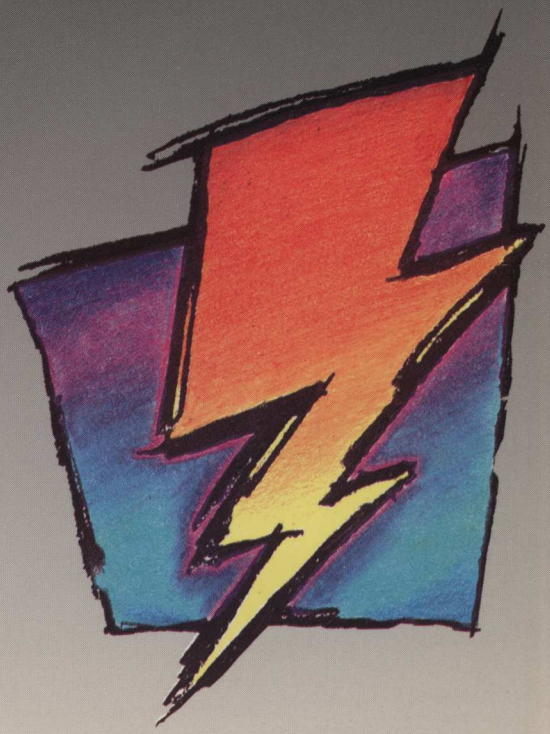
WATCOM C/386

User's Guide for PenPoint



WATCOM C/386

Optimizing Compiler and Tools

User's Guide for PenPoint

G. Coschi
J.B. Schueler
A.F. Scian

WATCOM Publications Limited
Waterloo, Ontario, Canada

Notice of Copyright

© Copyright 1991, by WATCOM Publications Limited.

All rights reserved. No part of this publication may be reproduced or used in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping or information storage and retrieval systems - without written permission of WATCOM Publications.

Disclaimer

WATCOM (The WATCOM Group Inc. and all of its subsidiaries) makes no representation or warranty with respect to the adequacy of this documentation or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall WATCOM, its employees, its contractors or the authors of this documentation be liable for special, direct, indirect or consequential damages, losses, costs, charges, claims, demands, or claim for lost profits, fees or expenses of any nature or kind.

ISBN 1-55094-034-1
Printed in Canada

WATCOM Support

Products and Technical Support

A broad range of software products, ongoing customer support services and software maintenance are available from WATCOM.

Bulletin Board Service

We offer our customers free access to a bulletin board service. Customers can access known problems and fixes available for the latest version of the software. The bulletin board can be used as a forum for developers to exchange ideas about their experiences with WATCOM software.

New Software and Updates

By constantly developing and upgrading software using new technology, WATCOM keeps users in the forefront.

Publications

Additional copies of this and related texts are available from WATCOM, which supplies a range of textbooks and teaching materials.

WATCOM	Telephone:	(519) 886-3700
415 Phillip Street	FAX:	(519) 747-4971
Waterloo, Ontario	BBS:	(519) 884-2103
CANADA N2L 3X2		

Preface

WATCOM C/386 is an implementation of the American National Standard for Information Systems - Programming Language C, ANSI X3.159-1989. The standard was developed by the ANSI X3J11 Technical Committee on the C Programming Language. In addition to the full C language standard, the compiler supports numerous extensions for the IBM PC environment.

WATCOM is well known for its language processors having developed, over the last decade, compilers and interpreters for the APL, BASIC, COBOL, FORTRAN and Pascal programming languages. From the start, WATCOM has been committed to developing portable software products. These products have been implemented on a variety of processor architectures including the IBM 370, the Intel 8086 family, the Motorola 6809 and 68000, the MOS 6502, and the Digital PDP11 and VAX. In most cases, the tools necessary for porting to these environments had to be created first. Invariably, a code generator had to be written. Assemblers, linkers and debuggers had to be created when none were available or when existing ones were inadequate.

Over the years, much research has gone into developing the "ultimate" code generator for the Intel 8086 family. We have continually looked for new ways to improve the quality of the emitted code, never being quite satisfied with the results. Several major revisions, including some entirely new approaches to code generation, have ensued over the years. Our latest version employs state of the art techniques to produce very high quality code for the 8086 family. We are now satisfied that we have a C software development system that will be of major benefit to those developing applications in C for the IBM PC and compatibles.

The *WATCOM C/386 Optimizing Compiler and Tools User's Guide for PenPoint* describes how to use WATCOM C/386 and related tools on the IBM PC and PS/2 to build PenPoint applications. The User's Guide is composed of four major parts plus appendices. These are *WATCOM C/386 Compiler Guide, WATCOM Linker, WATCOM Tools,* and *Advanced Topics.* The "Compiler Guide" introduces the Compile and Link utility and the WATCOM C/386 compiler. The "Linker" part of the User's Guide describes the use of the WATCOM Linker for PenPoint applications. The "Tools" part of the User's Guide describes WATCOM's library manager, strip utility, make and touch utilities, object file disassembler, and patch utility. The "Advanced Topics" section contains a brief overview of the Intel 80386 architecture, a description of the memory models supported by WATCOM C/386, the WATCOM C/386 calling conventions, and WATCOM C/386 pragmas.

The appendices contain information on the use of DOS or OS/2 environment variables by the various components of the package, a list of WATCOM C/386 diagnostic messages with an explanation of each, some commonly asked questions and answers to them, and the procedure for reporting a software problem.

In addition to this manual, we also provide the following documentation.

The *WATCOM C Language Reference* describes the ANSI C programming language and extensions to it which are supported by WATCOM C.

The *WATCOM C Library Reference for PenPoint* describes the WATCOM C Library. In this manual, we describe all of the WATCOM C Library routines.

Additional copies of this documentation may be ordered from:

WATCOM
415 Phillip St.
Waterloo, Ontario
CANADA N2L 3X2

This book was produced with the WATCOM GML electronic publishing system, a software tool developed and sold by WATCOM. In this system, writers use an ASCII text editor such as the WATCOM Editor to create source files containing text annotated with tags. These tags label the structural elements of the document, such as chapters, sections, paragraphs, and lists. The WATCOM GML software, which runs on the MS DOS, IBM VM/SP CMS, and Digital VAX/VMS operating systems, interprets the tags to format the text into a form such as you see here. Writers can produce output for a variety of printers, including laser printers, using separately specified layout directives for such things as font selection, column width and height, number of columns, etc. The result is type-set quality copy containing integrated text and graphics.

Acknowledgements

Many members of WATCOM Systems Inc. have made a significant contribution to the design and implementation of the WATCOM C/386 compiler, the code generator, and the many related tools. The design of this software is based upon ideas evolved and proven over the past decade in other software projects in which these people have been involved. The major project members of the WATCOM C/386 compiler and tools project have been Scott Beech, Geno Coschi, Fred Crigger, John Dahms, Craig Eisler, Steven Fry, Dean Gaudet, Michael Glaum, Jim Graham, Bill Hillis, Tom Ivey, Alex Kachura, Carmel Lau, Jim Randall, Jack Schueler, Anthony Scian, Brian Stecher, Aaron Stein, Anderson Wai, David Wellstood and David Yach.

Special Acknowledgements

The Plum Hall Validation Suite has been invaluable in verifying the conformance of the WATCOM C/386 compiler to the ANSI C Language Standard.

Many users have provided valuable feedback on preliminary versions of the WATCOM C/386 compiler and related tools. We would especially like to thank the following people for their input:

Eric Christensen, Fox Software
David Reed, Lotus Development Corp.
Richard Smith, Phar Lap Software, Inc.

Their comments were greatly appreciated. If you find problems in the documentation or have some good suggestions, we would like to hear from you. The last appendix in this guide describes various ways of communicating your ideas to WATCOM.

G. Coschi
J. B. Schueler
A. F. Scian

December, 1991.

Trademarks Used in this Manual

AutoCAD Development System is a trademark of Autodesk, Inc.

DOS/4G is a trademark of Rational Systems, Inc.

High C is a trademark of MetaWare, Inc.

IBM is a registered trademark of International Business Machines Corp.

Intel, 386, 486, C Code Builder are registered trademarks of Intel Corp.

Microsoft is a registered trademark of Microsoft Corp. Windows is a trademark of Microsoft Corp.

Phar Lap and 386|DOS-Extender are trademarks of Phar Lap Software, Inc.

PenPoint is a trademark of GO Corporation.

QNX is a registered trademark of Quantum Software Systems Ltd.

UNIX is a registered trademark of AT&T.

WATCOM and Express C are trademarks of WATCOM Systems, Inc.

Table of Contents

WATCOM C/386 Compiler Guide 1

1 Introduction 3

2 The WATCOM C/386 Compile and Link Utility 5
- 2.1 Introduction 5
- 2.2 WCL386 Command Line Format 5
- 2.3 WCL386 Options Summary 6
- 2.4 WCL386 Environment Variable 9
- 2.5 WCL386 Command Line Examples 9

3 The WATCOM C/386 Compiler 13
- 3.1 Introduction 13
- 3.2 WATCOM C/386 Command Line Format 13
- 3.3 WATCOM C/386 Options Summary 14
- 3.4 Compiler Options 16
- 3.5 WCC386 Environment Variable 37
- 3.6 WATCOM C/386 Command Line Examples 38
- 3.7 Compiler Diagnostics 40
- 3.8 WATCOM C/386 #include File Processing 42
- 3.9 WATCOM C/386 Preprocessor 44
- 3.10 WATCOM C Predefined Macros 45
- 3.11 WATCOM C/386 Extended Keywords 47
- 3.12 The WATCOM Code Generator 51
- 3.13 The WATCOM C/386 for PenPoint 1.0 Libraries 53

WATCOM Linker 55

4 The PenPoint Executable and DLL File Formats 57
- 4.1 The WATCOM Linker Command Line 57
- 4.2 Dynamic Link Libraries 61
  - 4.2.1 Creating Dynamic Link Libraries 62
  - 4.2.2 Using Dynamic Link Libraries 62
- 4.3 WATCOM Linker Directives 63
  - 4.3.1 The DEBUG Directive 64
    - 4.3.1.1 Line Numbering Information - DEBUG LINES 65
    - 4.3.1.2 Local Symbol Information - DEBUG LOCALS 65
    - 4.3.1.3 Typing Information - DEBUG TYPES 66
    - 4.3.1.4 All Debugging Information - DEBUG ALL 66
    - 4.3.1.5 Global Symbol Information 67
    - 4.3.1.6 Using DEBUG Directives 67

Table of Contents

4.3.1.7 Removing Debugging Information from an Executable File 68
4.3.2 The EXPORT Directive 68
4.3.3 The FILE Directive 69
4.3.4 The FORMAT Directive 71
4.3.5 The IMPORT Directive 74
4.3.6 The LIBRARY Directive 75
4.3.6.1 Searching for Libraries Specified in Environment Variables 76
4.3.7 The LIBFILE Directive 78
4.3.8 The LIBPATH Directive 79
4.3.9 The MODTRACE Directive 80
4.3.10 The NAME Directive 81
4.3.11 The NEWSEGMENT Directive 82
4.3.12 The OPTION Directive 82
4.3.12.1 The ALIGNMENT Option 84
4.3.12.2 The CASEEXACT Option 84
4.3.12.3 The DESCRIPTION Option 85
4.3.12.4 The DOSSEG Option 85
4.3.12.5 The HEAPSIZE Option 86
4.3.12.6 The MANYAUTODATA Option 87
4.3.12.7 The MAP Option 87
4.3.12.8 The MODNAME Option 88
4.3.12.9 The NAMELEN Option 88
4.3.12.10 The NEWFILES Option 89
4.3.12.11 The NOAUTODATA Option 89
4.3.12.12 The NODEFAULTLIBS Option 89
4.3.12.13 The OFFSET Option 90
4.3.12.14 The OLDLIBRARY Option 91
4.3.12.15 The ONEAUTODATA Option 91
4.3.12.16 The PACKCODE Option 91
4.3.12.17 The PROTMODE Option 92
4.3.12.18 The QUIET Option 93
4.3.12.19 The STACK Option 93
4.3.12.20 The STUB Option 94
4.3.12.21 The SYMFILE Option 94
4.3.12.22 The UNDEFSOK Option 95
4.3.12.23 The VERBOSE Option 96
4.3.12.24 The VERSION Option 96
4.3.13 The PATH Directive 97
4.3.14 The SEGMENT Directive 98
4.3.15 The SYMTRACE Directive 100
4.3.16 The SYSTEM Directive 100
4.3.17 The # Directive 102

Table of Contents

4.3.18 The @ Directive 103
4.4 Memory Layout 105
4.5 The WATCOM Linker Memory Requirements 106
4.6 Converting Microsoft Response Files to WATCOM Linker Directive Files 106
4.7 WATCOM Linker Error Messages 107

WATCOM Tools 125

5 The WATCOM Library Manager 127
5.1 Introduction 127
5.2 The WATCOM Library Manager Command Line 127
5.3 Adding Modules to a Library File 129
5.4 Deleting Modules from a Library File 130
5.5 Replacing Modules in a Library File 130
5.6 Extracting a Module from a Library File 131
5.7 Creating Import Libraries 132
5.8 Creating Import Library Entries 133
5.9 Commands from a File 133
5.10 WATCOM Library Manager Options 134
5.10.1 Suppress Creation of Backup File - "b" Option 134
5.10.2 Case Sensitive Symbol Names - "c" Option 134
5.10.3 Specify Output Directory - "d" Option 134
5.10.4 Generating Imports from Dynamic Link Libraries - "i" Option 135
5.10.5 Creating a Listing File - "l" Option 135
5.10.6 Always Create a New Library - "n" Option 136
5.10.7 Specifying an Output File Name - "o" Option 136
5.10.8 Specifying a Library Record Size - "p" Option 137
5.10.9 Operate Quietly - "q" Option 137
5.10.10 Strip Line Number Records - "s" Option 137
5.10.11 Trim Module Name - "t" Option 138
5.10.12 Explode Library File - "x" Option 138
5.11 WATCOM Library Manager Error Messages 139

6 The WATCOM Strip Utility 143
6.1 Introduction 143
6.2 The WATCOM Strip Utility Command Line 143
6.3 WATCOM Strip Utility Error Messages 144

7 The Make Utility 147
7.1 Introduction 147
7.2 WATCOM Make Reference 147

Table of Contents

7.2.1 WATCOM Make Command Line Format 148
7.2.2 WATCOM Make Options Summary 148
7.2.3 Command Line Options 149
7.2.4 Special Macros 153
7.3 Dependency Declarations 155
7.4 Multiple Dependents 156
7.5 Multiple Targets 156
7.6 Multiple Rules 157
7.7 Targets Without Any Dependents - ".SYMBOLIC" 159
7.8 Preserving Targets - ".PRECIOUS" 161
7.9 Ignoring Return Codes - ".IGNORE" 162
7.10 Erasing Targets After Error - ".ERASE" 163
7.11 Preserving Targets After Error - ".HOLD" 163
7.12 Suppressing Terminal Output - ".SILENT" 164
7.13 Macros 164
7.14 Implicit Rules 174
7.15 Double Colon Explicit Rules 185
7.16 Preprocessing Directives 187
7.16.1 File Inclusion 187
7.16.2 Conditional Processing 191
7.17 Command List Directives 196
7.18 MAKEINIT File 198
7.19 Command List Execution 198
7.20 Compatibility Between WATCOM Make and UNIX Make 204

8 The Touch Utility 207
8.1 WTOUCH Operation 208

9 The Object File Disassembler 209
9.1 Introduction 209
9.2 The Listing Option - "l[=<list_file>]" 210
9.3 The Source Option - "s[=<source_file>]" 211
9.4 The Assembly Format Option - "a", "au" 212
9.5 The Uppercase Opcode Option - "u" 212
9.6 The Uppercase Register Option - "r" 212
9.7 The Alternate Addressing Form Option - "b" 212
9.8 The External Symbols Option - "e" 213
9.9 The Public Symbols Option - "p" 213
9.10 Identifying Code Segments Option - "c=<code_name>" 214
9.11 An Example 215

10 The WATCOM Patch Utility 219

Table of Contents

10.1 Applying a Patch 219
10.2 Diagnostic Messages 220

Advanced Topics 223

11 80386 Architecture 225
11.1 Registers 225
11.2 General Registers 225
11.2.1 The EAX Register 227
11.2.2 The EBX Register 227
11.2.3 The ECX Register 227
11.2.4 The EDX Register 227
11.2.5 The ESP and EBP Registers 227
11.2.6 The ESI and EDI Registers 228
11.3 Segment Registers 228
11.3.1 The CS Register 229
11.3.2 The DS Register 229
11.3.3 The ES, FS and GS Registers 229
11.3.4 The SS Register 229
11.4 Instruction Pointer 229
11.5 Flags Register 229
11.5.1 The Virtual 8086 Mode 231
11.5.2 The Resume Flag 231
11.5.3 The Nested Task Flag 231
11.5.4 The I/O Privilege Level 231
11.5.5 The Carry Flag 231
11.5.6 The Parity Flag 231
11.5.7 The Auxiliary Carry Flag 231
11.5.8 The Zero Flag 232
11.5.9 The Sign Flag 232
11.5.10 The Trap Flag 232
11.5.11 The Interrupt Enable Flag 232
11.5.12 The Direction Flag 232
11.5.13 The Overflow Flag 233
11.6 Effective Address Calculation 233
11.6.1 Instruction Fetching 233
11.6.2 Data Fetching 234
11.6.2.1 Direct Addressing 234
11.6.2.2 Immediate Addressing 234
11.6.2.3 Base Addressing 234
11.6.2.4 Base + Displacement Addressing 234

Table of Contents

11.6.2.5 (Index * Scale) + Displacement Addressing 234
11.6.2.6 Base + Index + Displacement Addressing 235
11.6.2.7 Base + (Index * Scale) + Displacement Addressing 235

12 Pragmas 237
12.1 Introduction 237
12.2 Using Pragmas to Specify Options 239
12.3 Using Pragmas to Specify Default Libraries 240
12.4 Pack Pragmas 241
12.5 The ALLOC_TEXT Pragma 242
12.6 Auxiliary Pragmas 242
12.6.1 Specifying a Symbol's Attributes 242
12.6.2 Alias Names 243
12.6.3 Special Symbols 246
12.6.4 Alternate Names for Symbols 248
12.6.5 Describing Calling Information 249
12.6.5.1 Loading Data Segment Register 250
12.6.5.2 Forcing a Stack Frame 251
12.6.6 Describing Argument Information 252
12.6.6.1 Passing Arguments in Registers 252
12.6.6.2 Forcing Arguments into Specific Registers 255
12.6.6.3 Passing Arguments to In-Line Functions 256
12.6.6.4 Removing Arguments from the Stack 257
12.6.6.5 Passing Arguments in Reverse Order 257
12.6.7 Describing Function Return Information 258
12.6.7.1 Returning Function Values in Registers 259
12.6.7.2 Returning Structures 260
12.6.7.3 Returning Floating-Point Data 261
12.6.8 A Function that Never Returns 262
12.6.9 Describing How Functions Use Memory 263
12.6.10 Describing the Registers Modified by a Function 268
12.6.11 An Example 269
12.6.12 Auxiliary Pragmas and the 80x87 270
12.6.12.1 Using the 80x87 to Pass Arguments 270
12.6.12.2 Using the 80x87 to Return Function Values 274
12.6.12.3 Preserving 80x87 Floating-Point Registers Across Calls 275

Appendices 277

A. Use of Environment Variables 279

Table of Contents

B. WATCOM C/386 Diagnostic Messages 285
B.1 Warning Level 1 Messages 286
B.2 Warning Level 2 Messages 289
B.3 Warning Level 3 Messages 289
B.4 Error Messages 290
B.5 Informational Messages 307
B.6 errno Values and Their Meanings 309
B.7 Math Run-Time Error Messages 310

C. Commonly Asked Questions and Answers 313
C.1 Introduction 313
C.2 Why can't the compiler find "stdio.h"? 314
C.3 What do these undefined references mean? 315
C.4 Why aren't my variables set to zero? 316
C.5 How can I see my source files in the debugger? 317
C.6 What does "Stack Overflow!" mean? 321
C.7 Why do I get redefinition errors from WLINK? 323
C.8 How do I convert to WATCOM C/386? 325
C.8.1 Conversion from UNIX compilers 327
C.8.2 Conversion from IBM PC and PS/2 DOS compilers 328
C.9 What should I know about optimization? 330

D. Software Problem Reporting 333

List of Figures

Figure 1. Code Transformation 51
Figure 2. 80386 Registers 226
Figure 3. Segment Registers 228
Figure 4. Flags Register 230

WATCOM C/386 Compiler Guide

1 Introduction

This guide describes the following topics:

- An Introduction to WATCOM C/386 for PenPoint 1.0
- Creating Applications for PenPoint 1.0
- The WATCOM C/386 Compile and Link Utility
- The WATCOM C/386 Compiler and Libraries
- Linking PenPoint Applications
- The Librarian
- The Strip Utility
- The Make Utility
- The Touch Utility
- The Object File Disassembler
- The WATCOM Patch Utility
- The 80386 Architecture
- The Use of Compiler Pragmas

In addition to the above topics, the appendices cover the following areas.

- The Use of Environment Variables
- WATCOM C/386 Diagnostic Messages
- Commonly Asked Questions and Answers
- Software Problem Reporting

For information about PenPoint programming, see the PenPoint SDK (Software Development Kit) documentation.

2 The WATCOM C/386 Compile and Link Utility

2.1 Introduction

The WATCOM Compile and Link Utility is designed for generating applications, simply and quickly, using a single command line. It is not likely that you will use this utility for generating PenPoint 1.0 applications. However, this tool is part of the PenPoint 1.0 SDK.

2.2 WCL386 Command Line Format

The format of the WCL386 command line is:

WCL386 [files] [options]

The files and options may be specified in any order. WCL386 uses the extension of the file name to determine if it is a source file, an object file, or a library file. Files with extensions of ".OBJ" and ".LIB" are assumed to be object files and library files respectively. Files with any other extension, including none at all, are assumed to be C source files and will be compiled. If no file extension is specified for a file name then "C" is assumed. Pattern matching characters ("*" and "?") may be used in the file specifications.

Options are specified as starting with a slash (/) or a dash (–) and may be specified in any order. Options can include any of the WATCOM C/386 compiler options plus some additional options specific to WCL386. WATCOM C/386 compiler options are described in detail in the next chapter. A summary of WCL386 options is displayed on the screen by simply entering the "WCL386" command with no arguments.

2.3 WCL386 Options Summary

General options:	*Description:*
/c	compile the files only, do not link them
/p	use the protected-mode version of the compiler
/y	ignore the WCL386 environment variable (see below)

Compiler options:	*Description:*
/bd	build dynamic link library
/bm	build multi-threaded application
/bw	build default windowing application
/d1	include line number debugging information
/d1+	include line number debugging information plus typing information for global symbols and local structs and arrays
/d2	include full symbolic debugging information
/d2~	include full symbolic debugging information except for type information on typedef names
/d<name>[=text]	preprocessor #define name [text]
/e<number>	set error limit number (default /e20)
/ei	force enums to be of type 'int'
/en	emit routine name before prologue
/ez†	generate Phar Lap Easy OMF-386 object file (1)
/fi=<file_name>	force file_name to be included
/fo=<file_name>	set object or preprocessor output filename
/fpc	generate calls to floating-point library
/fpi†	generate inline 80387 instructions with emulation (default)
/fpi87†	generate inline 80387 instructions (equivalent to "7" option below)
/fpi287†	generate inline 80287 instructions
/g=<codegroup>	set code group name
/i=<directory>	add directory to list of include directories
/j	change char default from unsigned to signed
/mf	
/m{s,m,c,l}†	memory model (mf=flat, ms=small, mm=medium, mc=compact, ml=large) (default /mf)
/nc=<code_class_name>	set name of the code class
/nd=<data_seg_name>	set name of the "data" segment
/nm=<module_name>	set module name different from filename
/nt=<text_seg_name>	set name of the "text" segment

1 *Any options marked with a "†" must not be used for PenPoint application development.*

/o{a,c,d,e,f,f+,i,l,m,r,s,t,x}	control optimization
/3r†	generate 386 instructions based on 80386 instruction timings and use register-based argument passing conventions
/3s	generate 386 instructions based on 80386 instruction timings and use stack-based argument passing conventions
/4r†	generate 386 instructions based on 80486 instruction timings and use register-based argument passing conventions (default)
/4s	generate 386 instructions based on 80486 instruction timings and use stack-based argument passing conventions
/7†	generate inline 80x87 instructions
/p{cl}	preprocess file only, sending output to standard output ("c" include comments, "l" include #line directives)
/r†	save/restore segment registers
/s	remove stack overflow checks
/sg	stack growing
/u<name>	preprocessor #undef name
/v	output function declarations to .def file (with typedef names)
/w<number>	set warning level number (default /w1)
/we	treat all warnings as errors
/z{a,e}	disable/enable language extensions (default /ze)
/zc	place literal strings in code segment
/zd{f,p}†	allow DS register to "float" or "peg" it to DGROUP (default /zdp)
/zf{f,p}†	allow FS register to be used (default for all but flat memory model) or not be used (default for flat memory model)
/zg	output function declarations to .def (without typedef names)
/zk{0,1,2}	double-byte char support: 0=Kanji, 1=Chinese/Taiwanese, 2=Korean
/zl	suppress generation of library file names and references in object file
/zm	generate individual functions into separate segments
/zp[{1,2,4,8}]	set minimal structure packing (member alignment) (default /zp1)
/zq	operate quietly
/zs	syntax check only
/zt<size>	set data threshold (default /zt32767)
/zu†	do not assume that SS contains segment of DGROUP
/zW†	Microsoft Windows optimized prologue/epilogue code sequences (2)

2 *Any options marked with a "†" must not be used for PenPoint application development.*

Linker options:	*Description:*
/k<stack_size>	set stack size
/fd[=<directive_file>]	keep directive file and, optionally, rename it (default name is "__WCL__.LNK").
/fe=<executable>	name executable file
/fm[=<map_file>]	generate map file and name it (optional)
/l=<system_name>	link a program for the specified system. Among the supported systems are:

386 32-bit DOS executables (synonym for "PHARLAP" or "ERGO")

ADS 32-bit AutoCAD ADS executables

CODEBUILDER 32-bit Intel Code Builder executables

DOS4G 32-bit Rational Systems DOS Extender executables

EADI 32-bit AutoCAD ADI executables (emulation)

ERGO 32-bit Ergo DOS Extender executables

FADI 32-bit AutoCAD ADI executables (floating-point)

NOVELL 32-bit Novell Netware 386 NLMs

OS2V2 32-bit OS/2 V2.x executables

PENPOINT 32-bit PenPoint executables

PHARLAP 32-bit PharLap DOS Extender executables

WIN386 32-bit Windows executables

These names are among the systems identified in the WATCOM Linker initialization file, "WLINK.LNK". The WATCOM Linker "SYSTEM" directives, found in this file, are used to specify default link options for particular (operating) systems. Users can augment the WATCOM Linker initialization file with their own system definitions and these may be specified as an argument to the "l=" option. The "system_name" specified in the "l=" option is used to create a "SYSTEM system_name" WATCOM Linker directive when linking the application.

/x	make names case sensitive
@<directive_file>	include additional directive file
/"linker directives"	allows use of any linker directive

2.4 WCL386 Environment Variable

The **WCL386** environment variable can be used to specify commonly used ***WCL386*** options. These options are processed before options specified on the command line.

Example:

```
C>set wcl386=/d1 /ot
```

The above example defines the default options to be "d1" (include line number debugging information in the object file), and "ot" (favour time optimizations over size optimizations).

Whenever you wish to specify an option that requires the use of an "=" character, you can use the "#" character in its place. This is required by the syntax of the "SET" command.

Once the **WCL386** environment variable has been defined, those options listed become the default each time the ***WCL386*** command is used.

The **WCL386** environment variable is used by ***WCL386*** only. ***WCL386*** passes the relevant options to the WATCOM C/386 compiler. This environment variable is not examined by the WATCOM C/386 compiler.

> ***Hint:*** If you are running DOS and you use the same ***WCL386*** options all the time, you may find it handy to place the "SET WCL386" command in your DOS system initialization file, `AUTOEXEC.BAT`. If you are running OS/2 and you use the same ***WCL386*** options all the time, you may find it handy to place the "SET WCL386" command in your OS/2 system initialization file, `CONFIG.SYS`.

2.5 WCL386 Command Line Examples

For most small applications, the ***WCL386*** command will suffice. We have only scratched the surface in describing the capabilities of the ***WCL386*** command. The following examples describe the ***WCL386*** command in more detail.

Suppose that your application is contained in three files called `apdemo.c`, `aputils.c`, and `apdata.c`. We can compile and link all three files with one command.

Example 1:

```
C>wcl386 /d2 apdemo.c aputils.c apdata.c
```

The executable program will be stored in `apdemo.exp` since `apdemo` appeared first in the list. Each of the three files is compiled with the "d2" debug option. Debugging information is included in the executable file.

We can issue a simpler command if the current directory contains only our three C source files.

Example 2:

```
C>wcl386 /d2 *.c
```

WCL386 will locate all files with the ".c" filename extension and compile each of them. The name of the executable file will depend on which of the C source files is found first. Since this is a somewhat haphazard approach to naming the executable file, ***WCL386*** has an option, "fe", which will allow you to specify the name to be used.

Example 3:

```
C>wcl386 /d2 /fe=apdemo *.c
```

By using the "fe" option, the executable file will always be called `apdemo.exp` regardless of the order of the C source files in the directory.

If the directory contains other C source files which are not part of the application then other tricks may be used to identify a subset of the files to be compiled and linked.

Example 4:

```
C>wcl386 /d2 /fe=apdemo ap*.c
```

Here we compile only those C source files that begin with the letters "ap".

In our examples, we have recompiled all the source files each time. In general, we will only compile one of them and include the object code for the others.

Example 5:

```
C>wcl386 /d2 /fe=apdemo aputils.c ap*.obj
```

The source file `aputils.c` is recompiled and `apdemo.obj` and `apdata.obj` are included when linking the application. The ".obj" filename extension indicates that this file need not be compiled.

Example 6:

```
C>wcl386 /fe=demo *.c utility.obj
```

All of the C source files in the current directory are compiled and then linked with `utility.obj` to generate `demo.exp`.

Example 7:

```
C>set wcl386=/mf /d1
C>wcl386 grmain grutil graph.lib
```

The files `grmain.c` and `grutil.c` are compiled for the flat memory model and then linked with `graph.lib` to generate `grmain.exp`. Since the "d1" option is included in the **WCL386** environment string, line number debugging information will be included in the executable file.

Example 8:

```
C>set wcl386=/mf /d1 /ox /k4096
C>wcl386 /fe=grdemo gr*.c graph.lib /fd=grdemo
```

All C source files beginning with the letters "gr" are compiled and then linked with `graph.lib` to generate `grdemo.exp` which uses a 4K stack. The temporary linker directive file that is created by ***WCL386*** will be kept and renamed to `grdemo.lnk`.

For more complex applications, you should use the "Make" utility. See the chapter entitled the "The Make Utility" for a description of Make.

ext is the filename extension of the file to be compiled. If omitted, a filename extension of ".c" is assumed. If the period "." is specified but not the extension, the file is assumed to have no filename extension.

options is a list of valid WATCOM C/386 options, each preceded by a slash ("/") or a dash ("–"). Options may be specified in any order.

3.3 WATCOM C/386 Options Summary

In this section, we present a terse summary of the WATCOM C/386 options. The next section describes these options in more detail. This summary is displayed on the screen by simply entering the ***WCC386*** or ***WCC386P*** command with no arguments.

Compiler options:	*Description:*
/bd	build dynamic link library
/bm	build multi-threaded application
/bw	build default windowing application
/d1	include line number debugging information
/d1+	include line number debugging information plus typing information for global symbols and local structs and arrays
/d2	include full symbolic debugging information
/d2~	include full symbolic debugging information except for type information on typedef names
/d<name>[=text]	preprocessor #define name [text]
/e<number>	set error limit number (default /e20)
/ei	force enums to be of type 'int'
/en	emit routine name before prologue
/ez†	generate Phar Lap Easy OMF-386 object file (3)
/fi=<file_name>	force file_name to be included
/fo=<file_name>	set object or preprocessor output filename
/fpc	generate calls to floating-point library
/fpi†	generate inline 80387 instructions with emulation (default)
/fpi87†	generate inline 80387 instructions (equivalent to "7" option below)
/fpi287†	generate inline 80287 instructions
/g=<codegroup>	set code group name
/i=<directory>	add directory to list of include directories
/j	change char default from unsigned to signed
/mf	

3 *Any options marked with a "†" must not be used for PenPoint application development.*

/m{s,m,c,l}†	memory model (mf=flat, ms=small, mm=medium, mc=compact, ml=large) (default /mf)
/nc=<code_class_name>	set name of the code class
/nd=<data_seg_name>	set name of the "data" segment
/nm=<module_name>	set module name different from filename
/nt=<text_seg_name>	set name of the "text" segment
/o{a,c,d,e,f,f+,i,l,m,r,s,t,x}	control optimization
/3r†	generate 386 instructions based on 80386 instruction timings and use register-based argument passing conventions
/3s	generate 386 instructions based on 80386 instruction timings and use stack-based argument passing conventions
/4r†	generate 386 instructions based on 80486 instruction timings and use register-based argument passing conventions (default)
/4s	generate 386 instructions based on 80486 instruction timings and use stack-based argument passing conventions
/7†	generate inline 80x87 instructions
/p{cl}	preprocess file only, sending output to standard output ("c" include comments, "l" include #line directives)
/r†	save/restore segment registers
/s	remove stack overflow checks
/sg	stack growing
/u<name>	preprocessor #undef name
/v	output function declarations to .def file (with typedef names)
/w<number>	set warning level number (default /w1)
/we	treat all warnings as errors
/z{a,e}	disable/enable language extensions (default /ze)
/zc	place literal strings in code segment
/zd{f,p}†	allow DS register to "float" or "peg" it to DGROUP (default /zdp)
/zf{f,p}†	allow FS register to be used (default for all but flat memory model) or not be used (default for flat memory model)
/zg	output function declarations to .def (without typedef names)
/zk{0,1,2}	double-byte char support: 0=Kanji, 1=Chinese/Taiwanese, 2=Korean
/zl	suppress generation of library file names and references in object file
/zm	generate individual functions into separate segments
/zp[{1,2,4,8}]	set minimal structure packing (member alignment) (default /zp1)
/zq	operate quietly
/zs	syntax check only
/zt<size>	set data threshold (default /zt32767)
/zu†	do not assume that SS contains segment of DGROUP
/zW†	Microsoft Windows optimized prologue/epilogue code sequences (4)

3.4 Compiler Options

/bd This option causes the compiler to imbed the appropriate DLL library name in the object file.

/bm This option causes the compiler to imbed the appropriate multi-thread library name in the object file.

/bw This option causes the compiler to import a special symbol so that the default windowsing library code is linked into your application.

/d1 Line number debugging information is included in the object file. This option provides additional information to the VIDEO debugger (at the expense of larger object files and executable files). Line numbers are handy when debugging your application with VIDEO at the source code level.

/d1+ Line number debugging information plus typing information for global symbols and local structs and arrays is included in the object file. Although global symbol information can be made available to the VIDEO debugger through a WATCOM Linker option, typing information for global symbols and local structs and arrays must be requested when the source file is compiled. This option provides additional information to the VIDEO debugger (at the expense of larger object files and executable files).

/d2 In addition to line number information, local symbol and data type information is included in the object file. Although global symbol information can be made available to the VIDEO debugger through a WATCOM Linker option, local symbol and typing information must be requested when the source file is compiled. This option provides additional information to VIDEO (at the expense of larger object files and executable files). By default, WATCOM C/386 will select the "od" level of optimization if "d2" is specified (see the description of the "od" option). However, it will make the debugging chore somewhat easier.

/d2~ This option is similar to "d2" except that type information on typedef names is omitted. This option is used to reduce the amount of symbolic debugging information in executables.

4 *Any options marked with a "†" must not be used for PenPoint application development.*

/d<name>[=text]

The specified definition is included as if it were part of the source code. Use of this option is equivalent to including the following line in your source code.

```
#define name [text]
```

Example:

```
/d_MODDATE="87/05/04"
```

The above example is equivalent to a line in the source file containing:

```
#define _MODDATE "87/05/04"
```

/e<number> WATCOM C/386 will stop compilation after reaching *<number>* errors. By default, WATCOM C/386 will stop compilation after 20 errors.

/ei By default, WATCOM C/386 will allocate the smallest storage unit required to hold all possible values given for an enumerated list. This option can be used to force the compiler to allocate an 'int' for all enumerated types.

/en WATCOM C/386 will emit the function name into the object code as a string of characters just before the function prologue sequence is generated. The string is terminated by a byte count of the number of characters in the string.

```
; void Toaster( int arg )

        db      "Toaster", 7
public  Toaster
Toaster label   byte
        .
        .
        .
        ret
```

This option is intended for developers of embedded systems (ROM-based applications).

/ez† WATCOM C/386 will generate an object file in Phar Lap Easy OMF-386 (object module format) instead of the default Microsoft OMF. (5)

/fi=<file_name>

The specified file is included as if a

5 *Any options marked with a "†" must not be used for PenPoint application development.*

```
#include "<file_name>"
```

directive were placed at the start of the source file.

Example:

```
C>wcc386 report /fi=\penpoint\sdk\inc\common.cnv
```

/fo=<file_name>

Without the "p{cl}" option: The "fo" option is used to specify the default object file drive, path, file name and extension. If the object file name is not specified, it is constructed from the source file name. If the object file extension is not specified, it is ".obj" by default.

Example:

```
C>wcc386 report /fo=d:\proj\obj\
```

A trailing "\" must be specified for directory names. If, for example, the option was specified as

```
/fo=d:\proj\obj
```

then the object file would be called `d:\proj\obj.obj`.

A default filename extension must be preceded by a period (".").

Example:

```
C>wcc386 report /fo=d:\proj\obj\.dbo
```

With the "p{cl}" option: The preprocessor is selected using a form of the "p" option. The "fo" option is used to specify the default output file drive, path, file name and extension. If the output file name is not specified, it is constructed from the source file name. If the output file extension is not specified, it is ".i" by default.

Example:

```
C>wcc386 report /p /fo=d:\proj\prep\
```

A trailing "\" must be specified for directory names. If, for example, the option was specified as

```
/fo=d:\proj\prep
```

then the output file would be called `d:\proj\prep.i`. A default filename extension must be preceded by a period (".").

Example:

```
C>wcc386 report /p /fo=d:\proj\prep\.cpr
```

/fpc All floating-point arithmetic is done with calls to a floating-point emulation library. This option should be used for any of the following reasons:

1. Speed of floating-point emulation is favoured over code size.
2. An application containing floating-point operations is to be stored in ROM and an 80x87 will not be present in the system.

/fpi† WATCOM C/386 will generate in-line 80387 numeric data processor instructions into the object code for floating-point operations. When any module containing floating-point operations is compiled with the "fpi" option, coprocessor emulation software will be included in the application when it is linked. Thus, a math coprocessor need not be present at run-time. This is the default floating-point option if none is specified. The "fpi" option causes the macro ***__FPI__*** to be predefined by the compiler.

/fpi87† WATCOM C/386 will generate in-line 80387 numeric data processor instructions into the object code for floating-point operations. When the "fpi87" option is used exclusively, coprocessor emulation software is not included in the application when it is linked. An 80387 coprocessor (or 80486 processor) must be present at run-time. The "fpi87" option causes the macro ***__FPI__*** to be predefined by the compiler.

/fpi287† WATCOM C/386 will generate in-line 80287 numeric data processor instructions into the object code for floating-point operations. Use this option if you wish to support those few 386 systems that are equipped with an 80287 numeric data processor. When the "fpi287" option is used exclusively, coprocessor emulation software is not included in the application when it is linked. An 80287 coprocessor (or 80387 or 80486 processor) must be present at run-time. The "fpi287" option causes the macro ***__FPI__*** to be predefined by the compiler.

Notes:

1. When any module in an application is compiled with the "fpc" option, then all modules must be compiled with the "fpc" option.

2. When any module in an application is compiled with a particular "floating-point" option, then all modules must be compiled with the same option.

3. If you wish to have floating-point emulation software included in the application, you should select the "fpi" option. A math coprocessor need not be present at run-time.

4. Different math libraries are provided for applications which have been compiled with a particular floating-point option. See the section entitled "WATCOM C/386 Libraries" near the end of this chapter.

/g=<codegroup>

The generated code is placed in the group called "<codegroup>". The default "text" segment name will be "<codegroup>_TEXT" but this can be overridden by the "nt" option.

Example:

```
C>wcc386 report /g=RPTGROUP /s
```

/i=<directory>

where "<directory>" takes the form

```
[d:]path;[d:]path...
```

The specified paths are added to the list of directories in which WATCOM C/386 will search for "include" files. See the section entitled "WATCOM C/386 #include File Processing" for information on directory searching.

/j

The default ***char*** type is changed from an unsigned to a signed quantity. The "j" option causes the macro ***__CHAR_SIGNED__*** to be predefined by the compiler.

/m?

? is one of *f, s, m, c* or *l*. A particular memory model is selected when generating object code. (Note: In several places in the text, a pair of underscore characters appears as __ which resembles a single, elongated underscore.)

/mf

The "flat" memory model (code and data up to 4 gigabytes) is selected. By default, WATCOM C/386 will select this memory model. The following macro will be predefined.

__FLAT__

/ms†

The "small" memory model (small code, small data) is selected. The following macros will be predefined.

M_I86SM
__SMALL__

/mm† The "medium" memory model (big code, small data) is selected. The following macros will be predefined.

M_I86MM
__MEDIUM__

/mc† The "compact" memory model (small code, big data) is selected. The following macros will be predefined.

M_I86CM
__COMPACT__

/ml† The "large" memory model (big code, big data) is selected. The following macros will be predefined.

M_I86LM
__LARGE__

Memory models are described in detail in the chapter entitled "Memory Models". Other architectural aspects of the 80386 such as pointer size are discussed in the section entitled "Sizes of Predefined Types" in the chapter entitled "Assembly Language Considerations".

/nc=<code_class_name>

The default "code" class name is "CODE". The small code model "_TEXT" segment and the large code model "module_name_TEXT" segments belong to the "CODE" class. This option allows you to select a different class name for these code segments. The name of the "code" class is explicitly set to "code_class_name".

Note that the default "data" class names are "DATA" (for the "CONST", "CONST2" and "_DATA" segments) and "BSS" (for the "_BSS" segment). There is no provision for changing the data class names.

/nd=<prefix>

This option permits you to define a special prefix for the "CONST", "CONST2", "_DATA", and "_BSS" segment names. The name of the group to which these segments belong is also changed from "DGROUP" to "prefix_GROUP". This option is especially useful in the creation of Dynamic Link Library (DLL) routines.

Example:

```
C>wcc386 report /nd=spec
```

In the above example, the segment names become "specCONST", "specCONST2", "spec_DATA", and "spec_BSS" and the group name becomes "spec_GROUP".

By default, the data group "DGROUP" consists of the "CONST", "CONST2", "_DATA", and "_BSS" segments. The compiler places certain types of data in each segment. The "CONST" segment contains constant literals that appear in your source code.

Example:

```
char *birds[ 3 ] = { "robin", "finch", "wren" };

printf( "Hello world\n" );
```

In the above example, the strings "Hello world\n", "robin", "finch", etc. appear in the "CONST" segment.

The "CONST2" segment contains initialized read-only and read/write data such as scalars, structures, or arrays.

Example:

```
const int cvar = 1;
int var = 2;
int table[ 5 ] = { 1, 2, 3, 4, 5 };
char *birds[ 3 ] = { "robin", "finch", "wren" };
```

In the above example, the array `birds` is in the "CONST2" segment. The pointers in this array point to the strings "robin", "finch", etc. that appear in the "CONST" segment.

The "_BSS" segment contains uninitialized data such as scalars, structures, or arrays.

Example:

```
int var1;
int array1[ 400 ];
```

Other data segments containing data, specifically declared to be ***far*** or exceeding the data threshold (see "zt" option), are named "module_nameN_DATA" where "N" is a number greater than 5.

Example:

```
int far array2[400];
```

In the above example, `array2` is placed in the segment "report6_DATA" provided that the module name is "report".

/nm=<module_name>

By default, the object file name and the module name that is placed within it are constructed from the source file name. When the "nm" option is used, the module name that is placed into the object file is "module_name". For large code models, the "text" segment name is "module_name_TEXT" unless the "nt" option is used.

In the following example, the preprocessed output from `report.c` is stored on the ram disk "D" under the name `temp.c`. The file is compiled with the "nm" option so that the module name imbedded into the object file is "REPORT" rather than "TEMP".

Example:

```
C>wcc386 report /pl /fo=d:\temp.c
C>wcc386 d:\temp /nm=report /fo=report
```

Since the "fo" option is also used, the resultant object file is called `report.obj`.

/nt=<text_seg_name>

The name of the "text" segment is explicitly set to "text_seg_name". By default, the "text" segment name is "_TEXT" for small code models and "module_name_TEXT" for large code models.

```
Memory        Code
Model         Segment
-------       -------------------
flat          _TEXT
small         _TEXT
medium        module_name_TEXT
compact       _TEXT
large         module_name_TEXT
```

/o? *?* is any of *a, c, d, e, f, f+, i, l, m, r, s, t* or *x*. A particular code generation strategy is selected. The option letters may be strung together as in "/oailt".

/oa Alias checking is relaxed. When this option is specified, the code optimizer will assume that global variables are not indirectly referenced through pointers. This assumption may reduce the size of the code that is generated. The following example helps to illustrate this point.

Example:

```
extern int i;

void rtn( int *pi ) {

    int k;
    for( k = 0; k < 10; ++k ) {
        (*pi)++;
        i++;
    }
}
```

In the above example, if "i" and "*pi" referenced the same integer object then "i" would be incremented by 2 each time through the "for" loop and we would call the pointer reference "*pi" an alias for the variable "i". In this situation, the compiler could not bind the variable "i" to a register without making sure that the "in-memory" copy of "i" was kept up-to-date. In most cases, the above situation does not arise. Rarely would we reference the same variable directly by name and indirectly through a pointer in the same routine. The "oa" option instructs the code generator that such cases do not arise in the module to be compiled. The code generator will be able to produce more efficient code when it does not have to worry about the alias "problem".

/oc This option may be used to disable the optimization where a "CALL" followed by a "RET" (return) is changed into a "JMP" (jump) instruction. This option is required if you wish to link an overlayed program using the Microsoft DOS Overlay Linker. The Microsoft DOS Overlay Linker will create overlay calls for a "CALL" instruction only. This option is not required when using the WATCOM Linker.

/od Non-optimized code sequences are generated. The resulting code will be much easier to debug when using VIDEO. By default, WATCOM C/386 will select "od" if "d2" is specified. If "d2" is followed by one of the other "o?" options then "od" is overridden.

Example:

```
C>wcc386 report /d2 /os
```

/oe=<number_of_quads> Certain user functions are expanded inline. The criteria for which functions are selected for inline expansion is based on the "size" of the function in terms of the number of "quads" generated by the function. The number of "quads" generated corresponds closely with the number of operators used in an expression. Functions which require more than "<number_of_quads>" are not expanded inline. The default number is 20. This optimization is useful when locally-referenced functions are small in size.

Example:

```
C>wcc386 dhrystone /oe
```

/of This option selects the generation of traceable stack frames for those functions that contain calls or require stack frame setup. To use WATCOM's "Dynamic Overlay Manager" (DOS only), you must compile all modules using one of the "of" or "of+" options ("of" is sufficient). For near functions, the following function prologue sequence is generated.

```
push EBP
mov  EBP,ESP
```

For far functions, the following function prologue sequence is generated.

```
inc  EBP
push EBP
mov  EBP,ESP
```

The EBP value on the stack will be even or odd depending on the code model. For 16-bit DOS systems, the Dynamic Overlay Manager uses this information to determine if the return address on the stack is a short address (16-bit offset) or long address (32-bit segment:offset).

Example:

```
C>wcc386 toaster /of
```

/of+

This option selects the generation of traceable stack frames for all functions regardless of whether they contain calls or require stack frame setup. This option is intended for developers of embedded systems (ROM-based applications). To use WATCOM's "Dynamic Overlay Manager" (DOS only), you must compile all modules using one of the "of" or "of+" options ("of" is sufficient).

For near functions, the following function prologue sequence is generated.

```
push EBP
mov  EBP,ESP
```

For far functions, the following function prologue sequence is generated.

```
inc  EBP
push EBP
mov  EBP,ESP
```

The EBP value on the stack will be even or odd depending on the code model. For 16-bit DOS systems, the Dynamic Overlay Manager uses this information to determine if the return address on the stack is a short address (16-bit offset) or long address (32-bit segment:offset).

Example:

```
C>wcc386 toaster /of+
```

/oi Certain library functions are generated in-line. You must include the appropriate header file containing the prototype for the desired function so that it will be generated in-line. The functions that can be generated in-line are:

```
abs       _disable div       _enable   fabs
_fmemchr  _fmemcmp _fmemcpy  _fmemset  _fstrcat
_fstrcmp  _fstrcpy _fstrlen  labs      ldiv
_lrotl    _lrotr   inp       inpw      memchr
memcmp    memcpy   memset    movedata  outp
outpw     _rotl    _rotr     strcat    strchr
strcpy    strlen
```

The "oi" option causes the macro ***__INLINE_FUNCTIONS__*** to be predefined by the compiler.

/ol Loop optimizations are performed. This includes moving loop-invariant expressions outside the loops.

/om Generate inline 80x87 code for math functions like sin, cos, tan, etc. If this option is selected, it is the programmer's responsibility to make sure that arguments to these functions are within the range accepted by the `fsin`, `fcos`, etc. instructions since no run-time check is made.

If the "ot" option is also specified, the `exp` function is generated inline as well.

/or This option enables reordering of instructions (instruction scheduling) to achieve better performance on pipelined architectures such as the 486. Selecting this option will make it slightly more difficult to debug because the assembly language instructions generated for a source statement may be intermixed with instructions generated for surrounding statements.

/os Space is favoured over time when generating code (smaller code but possibly slower execution).

/ot Time is favoured over space when generating code (faster execution but possibly larger code).

/ox The "oilr" and "s" (no stack overflow checking) options are selected. When "ox" is combined with the "oa" and "ot" options, the code generator will attempt to give you the fastest executing code possible. Note that "/oaxt" is equivalent to "/oailrt /s".

By default, WATCOM C/386 selects a balance between "space" and "time".

/3r† WATCOM C/386 will generate 386 instructions based on 80386 instruction timings (see "4r" below).

- WATCOM C/386 will pass arguments in registers whenever possible. This is the default method used to pass arguments.
- All registers except EAX are preserved across function calls.
- When any form of the "fpi" option is used, the result of functions of type "float" and "double" is returned in ST(0)
- When the "fpc" option is used, the result of a function of type "float" is returned in EAX and the result of a function of type "double" is returned in EDX:EAX.
- The resulting code will be smaller than that which is generated for the stack-based method of passing arguments (see "3s" below).
- The default naming convention for all global functions is such that an underscore character ("_") is suffixed to the symbol name. The default naming convention for all global variables is such that an underscore character ("_") is prefixed to the symbol name.

/3s WATCOM C/386 will generate 386 instructions based on 80386 instruction timings (see "4s" below).

- WATCOM C/386 will pass all arguments on the stack.
- The EAX, ECX and EDX registers are not preserved across function calls.
- The FS and GS registers are not preserved across function calls.
- The result of a function of type "float" is returned in EAX. The result of a function of type "double" is returned in EDX:EAX.

- The resulting code will be larger than that which is generated for the register method of passing arguments (see "3r" above).

- The naming convention for all global functions and variables is modified such that no underscore characters ("_") are prefixed or suffixed to the symbol name.

These conventions are similar to those used by the MetaWare High C 386 compiler.

/4r† This option is identical to "3r" except that WATCOM C/386 will generate 386 instructions based on 80486 instruction timings. The code is optimized for 486 processors rather than 386 processors.

/4s This option is identical to "3s" except that WATCOM C/386 will generate 386 instructions based on 80486 instruction timings. The code is optimized for 486 processors rather than 386 processors.

/7† WATCOM C/386 will generate in-line 80387 numeric data processor instructions into the object code for floating-point operations. This option is equivalent to "fpi87".

/p{cl} The input file is preprocessed and, by default, is written to the standard output file. The "fo" option may be used to redirect the output to a file with default extension ".i". Specify "pc" if you wish to include the original source comments in the WATCOM C/386 preprocessor output file. Specify "pl" if you wish to include `#line` directives. Specify "pcl" or "plc" if you wish both source comments and #line directives.

The input file is preprocessed only. When "p" is specified, source comments and #line directives are not included. You must request these using the "c" and "l" suffixes. When the output of the preprocessor is fed into the WATCOM C/386 compiler, the `#line` directive enables the compiler to issue diagnostics in terms of line numbers of the original source file.

The options which are supported when the WATCOM C/386 preprocessor is requested are: "d", "fi", "fo", "i", "m?", and "u".

/r† This option instructs WATCOM C/386 to generate function prologue and epilogue sequences that save and restore any segment registers that are modified by the function. Caution should be exercised when using this option. If the value of the segment register being restored matches the value of a segment that was freed within the function, a general protection fault will occur in protected-mode environments. By default WATCOM C/386 does not save and

restore segment registers. This option is provided for compatibility with the version 8.0 release.

/s

Stack overflow checking is omitted from the generated code. By default, WATCOM C/386 will emit code at the beginning of every function that checks for the "stack overflow" condition. This option can be used to disable this feature. The "stack overflow check" run-time routine is called `__STK`.

/sg

This option is useful for OS/2 2.x multi-threaded applications. It requests the code generator to emit a run-time call at the start of any function that has more than 4K bytes of automatic variables (variables located on the stack). Under OS/2 2.x, the stack is grown automatically in 4K pages using the stack "guard page" mechanism. The stack consists of in-use committed pages topped off with a special guard page. A memory reference into the 4K guard page causes OS/2 to grow the stack by one 4K page and to add a new 4K guard page. This works fine when there is less than 4K of automatic variables in a function. When there is more than 4K of automatic data, the stack must be grown in an orderly fashion, 4K bytes at a time, until the stack has grown sufficiently to accommodate all the automatic variable storage requirements.

The "stack growth" run-time routine is called `__GRO`.

/u<identifier>

The "u" option may be used to turn off the definition of a predefined macro. If no identifier is specified then all predefined macros with the exception of the memory model macros are undefined.

Example:

```
C>wcc386 report /uM_I86
```

/v

WATCOM C/386 will output function declarations to a file with the same filename as the C source file but with extension ".def". The "definitions" file may be used as an "include" file when compiling other modules in order to take advantage of WATCOM C/386's function and argument type checking.

/w<number>

WATCOM C/386 will issue only warning type messages of severity *<number>* or below. Type 1 warning messages are the most severe while type 3 warning messages are the least severe.

/w0	no warning messages are issued
/w1	only severity type 1 warning messages are issued
/w2	both severity type 1 and 2 warning messages are issued
/w3	severity type 1, 2 and 3 warning messages are issued
/w4	severity type 1, 2 and 3 warning messages are issued, and causes informational messages to be logged to the error file

/we By default, WATCOM C/386 will continue to create an object file when there are warnings produced. This option can be used to treat all warnings as errors, thereby preventing the compiler from creating an object file if there are warnings found within a module.

/za This option helps to ensure that the module to be compiled conforms to the ANSI C programming language specification. The "za" option causes the macro ***NO_EXT_KEYS*** (no extended keywords) to be predefined by the compiler. See also the description of the "ze" option.

/zc The "zc" option causes the code generator to place literal strings and *const* items in the code segment.

Example:

```
const int cvar = 1;
int var = 2;
const int ctable[ 5 ] = { 1, 2, 3, 4, 5 };
char *birds[ 3 ] = { "robin", "finch", "wren" };
```

In the above example, `cvar` and `ctable` and the strings "robin", "finch", etc. are placed in the code segment. This option is supported in large data memory models only.

/zd{f,p}† The "zdf" option allows the code generator to use the DS register to point to other segments besides "DGROUP" (default in compact and large memory models). The "zdp" option informs the code generator that the DS register must always point to "DGROUP" (default in small, medium and flat memory models).

/ze The "ze" option (default) enables the use of the following WATCOM C/386 extensions:

1. The requirement for at least one external definition per module is relaxed.

2. Some forgiveable pointer type mismatches become warnings instead of errors.

3. Inline math functions are allowed (note that *errno* will not be set by inline functions).

4. Anonymous structs/unions are allowed.

   Example:
   ```
   struct {
       int a;
       union {
           int   b;
           float alt_b;
       };
       int c;
   } x;
   ```

   In the above example, "x.b" is a valid reference to the "b" field.

5. ANSI function prototype scope rules are relaxed to allow the following program to compile without any errors.

   Example:
   ```
   void foo( struct a *__p );

   struct a {
       int b;
       int c;
   };

   void bar( void )
   {
       struct a x;
       foo( &x );
   }
   ```

   According to a strict interpretation of the ANSI C standard, the function prototype introduces a new scope which is terminated at the semicolon (;). The effect of this is that the structure tag "a" in the function "foo" is not the same structure tag "a" defined after the

prototype. A diagnostic must be issued for a conforming ANSI C implementation.

6. A trailing comma (,) is allowed after the last constant in an enum declaration.

   Example:
   ```
   enum colour { RED, GREEN, BLUE, };
   ```

7. The ANSI requirement that all enums have a base type of *int* is relaxed. The motivation for this extension is conservation of storage. Many enums can be represented by integral types that are smaller in size than an *int*.

   Example:
   ```
   enum colour { RED, GREEN, BLUE, };

   void foo( void )
   {
       enum colour x;

       x = RED;
   }
   ```

   In the example, "x" can be stored in an *unsigned char* because its values span the range 0 to 2.

8. The ANSI requirement that the base type of a bitfield be *int* or *unsigned* is relaxed. This allows a programmer to allocate bitfields from smaller units of storage than an *int* (e.g., *unsigned char*).

   Example:
   ```
   struct {
       unsigned char a : 1;
       unsigned char b : 1;
       unsigned char c : 1;
   } x;

   struct {
       unsigned a : 1;
       unsigned b : 1;
       unsigned c : 1;
   } y;
   ```

In the above example, the size of "x" is the same size as an *unsigned char* whereas the size of "y" is the same size as an *unsigned int*.

9. The following macros are defined.

```
_near, near
_far, far
_huge, huge
_cdecl, cdecl
_pascal, pascal
_fortran, fortran
_interrupt, interrupt
_export
_loadds
_saveregs
```

See also the description of the "za" option.

/zf{f,p}† The "zff" option allows the code generator to use the FS register (default for all but flat memory model). The "zfp" option informs the code generator that the FS register must not be used (default in flat memory model).

/zg The "zg" option is similar to the "v" option except that function declarations will be output to the "DEF" file using base types (i.e., typedefs are reduced to their base type).

Example:

```
typedef unsigned int UINT;
UINT f( UINT x )
 {
    return( x + 1 );
 }
```

If you use the "v" option, the output will be:

```
extern UINT f(UINT );
```

If you use the "zg" option, the output will be:

```
extern unsigned int f(unsigned int );
```

/zk{0,1,2} This option causes the compiler to recognize double-byte characters in strings. When the compiler scans a text string enclosed in quotes ("), it will recognize the first byte of a double-byte character and suppress lexical analysis of the

second byte. This will prevent the compiler from misinterpreting the second byte as a "\" or quote (") character.

/zk, /zk0 These options cause the compiler to process strings for Japanese double-byte characters (range 0x81 - 0x9F and 0xE0 - 0xFC). The characters in the range A0 - DF are single-byte Hiragana.

/zk1 This option causes the compiler to process strings for Traditional Chinese and Taiwanese double-byte characters (range 0x81 - 0xFC).

/zk2 This option causes the compiler to process strings for Korean Hangeul double-byte characters (range 0x81 - 0xFD).

/zl By default, WATCOM C/386 places in the object file the names of the C libraries that correspond to the memory model and floating-point options that were selected. The WATCOM Linker uses these library names to select the libraries required to link the application. If you use the "zl" option, the library names will not be included in the generated object file.

The WATCOM C/386 compiler may generate external references for library code that conveniently cause the linker to link in different code. One such case is: if you have any functions that pass or return floating-point values (i.e., float or double), the compiler will insert an external reference that will cause the floating-point formatting routines to be included in the executable. The "zl" option will disable these external references.

Use this option when you wish to create a library of object modules which do not contain WATCOM C/386 library name references.

/zm The "zm" option instructs the code generator to place each function into a separate segment. This option is used in paging environments where special segment ordering may be employed. The "alloc_text" pragma is often used in conjunction with this option to place functions into specific segments. The disadvantages to this option are:

1. Static functions are "far called" in big code models. The "near call" optimization is lost.
2. The "common epilogue" optimization is lost.

/zp[{1,2,4,8}]

The "zp" option allows you to specify the alignment of members in a structure. The default is "zp1". The alignment of structure members is described in the following table. If the size of the member is 1, 2, 4 or 8, the alignment is given

for each of the "zp" options. If the member of the structure is an array or structure, the alignment is described by the row "x".

```
                        zp1      zp2      zp4      zp8
sizeof(member)   \-------------------------------
        1        |   0        0        0        0
        2        |   0        2        2        2
        4        |   0        2        4        4
        8        |   0        2        4        8
        x        |   aligned to largest member
```

An alignment of 0 means no alignment, 2 means word boundary, 4 means doubleword boundary, etc. If the largest member of structure "x" is 1 byte then "x" is not aligned. If the largest member of structure "x" is 2 bytes then "x" is aligned according to row 2. If the largest member of structure "x" is 4 bytes then "x" is aligned according to row 4. If the largest member of structure "x" is 8 bytes then "x" is aligned according to row 8.

To understand why structure member alignment may be important, consider the following example.

Example:

```
typedef struct memo_el {
    char           date[9];
    struct memo_el *prev,*next;
    ref_number     int;
} memo;
```

In the above example, the default alignment "zp1" will cause the pointer and integer items to be aligned on odd addresses since the array "date" is 9 bytes in length. On computer systems that have a 16-bit (or 32-bit) bus, improved performance can be obtained when these items are aligned on an even boundary.

/zq The "quiet mode" option causes the informational messages displayed by the WATCOM C/386 compiler to be suppressed. Normally, messages are displayed identifying the compiler and summarizing the number of lines compiled. As well, a dot is displayed every few seconds while the code generator is active, to indicate that the compiler is still working. These messages are all suppressed by the "quiet mode" option. Error and warning messages are not suppressed.

/zs WATCOM C/386 will check the source code only and omit the generation of object code. Syntax checking, type checking, and so on are performed as usual.

/zt<size> The "data threshold" option is used to set the minimum size for data objects to be included in the default data segment. This option can be used with the

compact and large memory models only where there can be more than one data segment. Normally, all data objects smaller than 32768 bytes in size are placed in a default data segment, unless they are specifically declared to be ***far*** items. When there is a large amount of static data, it is often useful to set the data threshold size so that all objects of this size or larger are placed into another segment. For example, the option:

```
/zt100
```

causes all data objects of 100 bytes or more to be implicitly declared as ***far*** and placed in their own data segment. The default data threshold value is 32767. If the "zt" option is specified without a size, the data threshold value is 256.

/zu† The "zu" option relaxes the restriction that the SS register contains the base address of the default data segment, "DGROUP". Normally, all data items are placed into the group "DGROUP" and the SS register contains the base address of this group. When the "zu" option is selected, the SS register is volatile (assumed to point to another segment) and any global data references require loading a segment register such as DS with the base address of "DGROUP".

This option is useful when compiling routines that are to be placed in a Dynamic Link Library (DLL) since the SS register points to the stack segment of the calling application upon entry to the function.

/zW† This option causes the compiler to generate the prologue/epilogue code sequences necessary for use in 32-bit Microsoft Windows applications.

The "zW" option causes the macro ***__WINDOWS_386__*** to be predefined by the compiler.

3.5 WCC386 Environment Variable

The **WCC386** environment variable can be used to specify commonly used WATCOM C/386 options. These options are processed before options specified on the command line.

Example:

```
C>set wcc386=/d1 /ot
```

The above example defines the default options to be "d1" (include line number debugging information in the object file), and "ot" (favour time optimizations over size optimizations).

Whenever you wish to specify an option that requires the use of an "=" character, you can use the "#" character in its place. This is required by the syntax of the "SET" command.

Once the **WCC386** environment variable has been defined, those options listed become the default each time the ***WCC386*** or ***WCC386P*** command is used.

The **WCC386** environment variable is used by ***WCC386*** and ***WCC386P*** only. The WATCOM C/386 compiler command line can be used to override any options specified in the environment string.

This environment variable is not examined by WCL386, the WATCOM Compile and Link Utility. Since WCL386 passes the relevant options found in the **WCL386** environment variable to the WATCOM C/386 compiler command line, the **WCL386** environment variable options take precedence over the **WCC386** environment variable options.

> ***Hint:*** If you are running DOS and you use the same compiler options all the time, you may find it handy to place the "SET WCC386" command in your DOS system initialization file, `AUTOEXEC.BAT`. If you are running OS/2 and you use the same WCC386 options all the time, you may find it handy to place the "SET WCC386" command in your OS/2 system initialization file, `CONFIG.SYS`.

3.6 WATCOM C/386 Command Line Examples

The following are some examples of using WATCOM C/386 to compile a C source program.

Example 1:

```
C>wcc386 report /d1 /s
```

The WATCOM C/386 compiler processes `report.c` producing an object file which contains source line number information. Stack overflow checking is omitted from the object code.

Example 2:

```
C>wcc386 /mf /3s calc
```

WATCOM C/386 compiles `calc.c` for the "flat" memory model using the stack-based argument passing convention. Memory models are described in the chapter entitled "Memory Models". Argument passing conventions are described in the chapter entitled "Assembly Language Considerations".

Example 3:

```
C>wcc386 kwikdraw /3r /fpi87 /oaimxt
```

The WATCOM C/386 compiler processes `kwikdraw.c` producing object code for an Intel 80386 system equipped with an Intel 80387 numeric data processor. The register-based argument passing convention is employed. The resulting code will be highly optimized for 80386/80486 systems.

Example 4:

```
C>wcc386 ..\source\modabs /d2
```

The WATCOM C/386 compiler processes `..\source\modabs.c` (a file in a directory which is adjacent to the current one). The object file is placed in the current directory. Included with the object code and data is information on local symbols and data types. The code generated is straight-forward, unoptimized code which can be readily debugged with the WATCOM debugger, VIDEO.

Example 5:

```
C>set wcc386=/i#\includes /mf
C>wcc386 \cprogs\grep.tst /fi=iomods.c
```

The WATCOM C/386 compiler processes the C program contained in the file `\cprogs\grep.tst`. The file `iomods.c` is included as if it formed part of the source input stream. The options `/i=\includes` and `/mf` are defaults each time the WATCOM C/386 compiler is invoked. The `/mf` option could be overridden on the command line. After looking for an "include" file in the current directory, WATCOM C/386 will search each directory listed in the "i" path. See the section entitled "WATCOM C/386 #include File Processing" for more information. The object code that is generated follows the "flat" memory model.

Example 6:

```
C>wcc386 grep /fo=..\obj\ /mf
```

The WATCOM C/386 compiler processes the C program contained in the file `grep.c` which is located in the current directory. The object file is placed in the directory `..\obj` under the name `grep.obj`. The object code that is generated follows the "flat" memory model.

Example 7:

```
C>wcc386 /dDBG=1 grep /fo=..\obj\.dbo /mf
```

The WATCOM C/386 compiler processes the C program contained in the file `grep.c` which is located in the current directory. The macro "DBG" is defined so that conditional debugging statements that have been placed in the source are compiled. The object file is placed in the directory `..\obj` and its filename extension will be ".dbo" (instead of ".obj"). Selection of a different filename extension permits easy identification of object files that have been compiled with debugging statements. The object code that is generated follows the "flat" memory model.

Example 8:

```
C>wcc386 /g=GKS /s \gks\gopks
```

The WATCOM C/386 compiler generates code for `gopks.c` and places it into the "GKS" group. If the "g" option had not been specified, the code would not have been placed in any group. Assume that this file contains the definition of the routine `gopengks` as follows:

```
void far gopengks( int workstation, long int h )
{
    .
    .
    .
}
```

For a small code model, the routine `gopengks` must be defined in this file as ***far*** since it is placed in another group. The "s" option is also specified to prevent a run-time call to the stack overflow check routine which will be placed in a different code segment at link time. The `gopengks` routine must be prototyped by C routines in other groups as

```
void far gopengks( int workstation, long int h );
```

since it will appear in a different code segment.

3.7 Compiler Diagnostics

If the WATCOM C/386 compiler prints diagnostic messages to the screen, it will also place a copy of these messages in a file in your current directory. The file will have the same file name as the source file and an extension of ".err". The compiler issues two types of diagnostic messages, namely warnings or errors. A warning message does not prevent the production of an object file. However, error messages indicate that a problem is severe enough that it must be corrected before the compiler will produce an object file. The error file is a handy reference when you wish to correct the errors in the source file.

Just to illustrate the diagnostic features of WATCOM C/386, we will modify the "hello" program (introduced in chapter 1) in such a way as to introduce some errors.

Example:

```
#include <stdio.h>

int main()
  {
     int x;
     printf( "Hello world\n" );
     return( y );
  }
```

In this example, we have added the line

```
int x;
```

and changed the line

```
return( 0 );
```

to read

```
return( y );
```

If we compile this program with the "warning" option,

```
C>wcc386 hello /w3
```

the following output appears on the screen.

```
hello.c(7): Error! E1011: Symbol 'y' has not been declared
hello.c(5): Warning! W202: Symbol 'x' has been defined, but not
                           referenced
hello.c: 8 lines, included 186, 1 warnings, 1 errors
```

Here we see an example of both types of messages. An error and a warning message have been issued. As indicated by the error message, we require a declarative statement for the identifier `y`. The warning message indicates that, while it is not a violation of the rules of C to define a variable without ever using it, we probably did not intend to do so. Upon examining the program, we find that:

1. the variable `x` should have been assigned a value, and
2. the variable `y` has probably been incorrectly typed and should have been entered as `x`.

The complete list of WATCOM C/386 diagnostic messages is presented in an appendix of this guide.

3.8 WATCOM C/386 #include File Processing

When using the `#include` preprocessor directive, a header is identified by a sequence of characters placed between the "<" and ">" delimiters (e.g., <file>) and a source file is identified by a sequence of characters enclosed by quotation marks (e.g., "file"). WATCOM C/386 makes a distinction between the use of "<>" or quotation marks to surround the name of the file to be included. The search techniques for header files and source files are slightly different. Consider the following example.

Example:

```
#include <stdio.h>   /* a header file */
#include "stdio.h"   /* a source file */
```

You should use "<" and ">" when referring to standard or system header files and quotation marks when referring to your own source and header files.

The character sequence placed between the delimiters in an `#include` directive represents the name of the file to be included. The file name may include drive, path, and extension.

It is not necessary to include the drive and path specifiers in the file specification when the file resides on a different drive or in a different directory. WATCOM C/386 provides a mechanism for looking up include files which may be located in various directories and disks of the computer system. When the drive and path are omitted from a file specification, WATCOM C/386 searches directories for header and source files in the following order:

1. For source files only (i.e., the name is surrounded by quotation marks), the current directory is searched.

2. For header and source files, each directory listed in the "i" path is searched (in the order that they were specified).

3. For header and source files, each directory listed in the **INCLUDE** environment variable is searched (in the order that they were specified);

4. For source files only (i.e., the name is surrounded by quotation marks), an adjacent "H" directory (i.e., `..\h`) is searched if it exists.

In the above example, `<stdio.h>` and `"stdio.h"` could refer to two different files if there is a `stdio.h` in the current directory and one in the WATCOM C/386 include file

directory (\PENPOINT\SDK\INC) and the current directory is not listed in an "i" path or the **INCLUDE** environment variable.

The compiler will search the directories listed in "i" paths (see description of the "i" option) and the **INCLUDE** environment variable in a manner analogous to that which DOS or OS/2 will use when searching for programs by using the **PATH** environment variable.

The DOS or OS/2 "SET" command is used to define an **INCLUDE** environment variable that contains a list of directories. A command of the form

```
SET INCLUDE=[d:]path;[d:]path...
```

is issued before running WATCOM C/386 the first time. The brackets indicate that the drive "d:" is optional and the ellipsis indicates that any number of paths may be specified.

We illustrate the use of the `#include` directive in the following example.

Example:

```
#include <stdio.h>
#include <time.h>
#include <dos.h>
#include "common.c"

int main()
  {
    initialize();
    update_files();
    create_report();
    finalize();
  }
#include "part1.c"
#include "part2.c"
```

If the above text is stored in the source file `report.c` in the current directory then we might issue the following commands to compile the application.

```
C>rem -- Two places to look for include files
C>set include=c:\penpoint\sdk\inc;b:\headers
C>rem -- Now compile application specifying a
C>rem    third location for include files
C>wcc386 report /fo=..\obj\ /i=..\source
```

In the above example, the DOS or OS/2 "SET" command is used to define the **INCLUDE** environment variable. It specifies that the `\penpoint\sdk\inc` directory (of the "C" disk) and the `\headers` directory (a directory of the "B" disk) are to be searched.

The WATCOM C/386 "i" option defines a third place to search for include files. The advantage of the **INCLUDE** environment variable is that it need not be specified each time the compiler is run.

3.9 WATCOM C/386 Preprocessor

The WATCOM C/386 preprocessor forms an integral part of WATCOM C/386. When any form of the "p" option is specified, only the preprocessor is invoked. No code is generated and no object file is produced. The output of the preprocessor is written to the standard output file, although it can also be redirected to a file using the "fo" option. Suppose the following C program is contained in the file `msgid.c`.

Example:

```
#define _IBMPC 0
#define _IBMPS2 1

#if _TARGET == _IBMPS2
char *SysId = { "IBM PS/2" };
#else
char *SysId = { "IBM PC" };
#endif

/* Return pointer to System Identification */

char *GetSysId()
  {
    return( SysId );
  }
```

We can use the WATCOM C/386 preprocessor to generate the C code that would actually be compiled by the WATCOM C/386 compiler by issuing the following command.

```
C>wcc386 msgid /plc /fo /d_TARGET=_IBMPS2
```

The file `msgid.i` will be created and will contain the following C code.

```
#line 1 "msgid.c"



char *SysId = { "IBM PS/2" };
#line 9 "msgid.c"

/* Return pointer to System Identification */

char *GetSysId()
  {
    return( SysId );
  }
```

Note that the file `msgid.i` can be used as input to the WATCOM C/386 compiler.

```
C>wcc386 msgid.i
```

Since `#line` directives are present in the file, the WATCOM C/386 compiler can issue error messages in terms of the original source file line numbers.

3.10 WATCOM C Predefined Macros

The WATCOM C compiler runs on various host operating systems. The following macros are defined for particular host operating systems. (Note: In several places in the following text, a pair of underscore characters appears as __ which resembles a single, elongated underscore.)

MSDOS	identifies that you are using the version of the WATCOM C compiler that runs under MS/DOS.
__OS2__	identifies that you are using the version of the WATCOM C compiler that runs under IBM OS/2.
__QNX__	identifies that you are using the version of the WATCOM C compiler that runs under QNX.

The WATCOM C compiler supports various target systems. The following macros are defined for 16-bit and 32-bit target systems.

M_I86 identifies the target as a 16-bit Intel 80x86 compatible environment (defined by the WATCOM C compiler).

__386__ identifies the target as a 32-bit Intel 80386 compatible environment. (defined by the WATCOM C/386 compiler).

In addition to the previous target defining macros, the following macros may be defined.

__NETWARE_386__ identifies the target system as Novell NetWare 386. This macro is defined by the Novell/WATCOM C SQL Compiler. If this macro is defined, the macro ***__386__*** will also be defined.

__WINDOWS__ identifies the target system as 16-bit Microsoft Windows. This macro is only defined when the WATCOM C "zw" or "zW" options are specified. If this macro is defined, the macro ***M_I86*** will also be defined.

__WINDOWS_386__ identifies the target system as 32-bit Microsoft Windows. This macro is only defined when the WATCOM C/386 "zW" option is specified. If this macro is defined, the macro ***__386__*** will also be defined.

The following macros indicate which compiler is compiling the C source code.

__WATCOMC__ The macro `__WATCOMC__` can be used to determine if the WATCOM C/386 compiler is compiling the program with a statement similar to:

```
#ifdef __WATCOMC__
    ...
#endif
```

The value of the macro depends on the version number of the compiler. The value is 100 times the version number (version 8.5 yields 850, version 9.0 yields 900, etc.).

__EXPRESSC__ The macro `__EXPRESSC__` can be used to determine if the WATCOM Express C compiler is compiling the program with a statement similar to:

```
#ifdef __EXPRESSC__
    ...
#endif
```

The Express C compiler runs on 16-bit DOS systems only.

The following macros are defined under the conditions described.

__FPI__ † is defined when any of the "fpi", "fpi87" or "fpi287" options are used. (6)

__CHAR_SIGNED__ is defined when the "j" option is used.

__INLINE_FUNCTIONS__ is defined when the "oi" option is used.

NO_EXT_KEYS is defined when the "za" option is used.

__FLAT__ is defined when the "mf" option is used.

M_I86SM, __SMALL__ † are defined when the "ms" option is used.

M_I86MM, __MEDIUM__ † are defined when the "mm" option is used.

M_I86CM, __COMPACT__ † are defined when the "mc" option is used.

M_I86LM, __LARGE__ † are defined when the "ml" option is used.

3.11 WATCOM C/386 Extended Keywords

WATCOM C/386 supports the use of some special keywords to describe system dependent attributes of functions and other object names. These attributes are inspired by the Intel processor architecture and the plethora of function calling conventions in use by compilers for this architecture. In keeping with the ANSI C language standard, WATCOM C/386 uses the double underscore ("_") prefix with these lowercase keywords. To support compatibility with

6 *Any macros marked with a "†" must not be used for PenPoint application development.*

other C compilers, alternate forms of these keywords are also supported through predefined macros.

__near †

WATCOM C/386 supports the ***__near*** keyword to describe functions and other object names that are in near memory and pointers to near objects.

WATCOM C/386 predefines the macros ***near*** and ***_near*** to be equivalent to the ***__near*** keyword. (7)

__far †

WATCOM C/386 supports the ***__far*** keyword to describe functions and other object names that are in far memory and pointers to far objects.

WATCOM C/386 predefines the macros ***far*** and ***_far*** to be equivalent to the ***__far*** keyword.

__huge †

WATCOM C/386 supports the ***__huge*** keyword to describe functions and other object names that are in huge memory and pointers to huge objects. WATCOM C/386 treats these as equivalent to far objects.

WATCOM C/386 predefines the macros ***huge*** and ***_huge*** to be equivalent to the ***__huge*** keyword.

__based

WATCOM C/386 supports the ***__based*** keyword to describe pointers to objects that appear in other segments or the objects themselves. See the "WATCOM C Language Reference" for an explanation of the ***__based*** keyword.

__segment

WATCOM C/386 supports the ***__segment*** keyword which is used when describing objects of type segment. See the "WATCOM C Language Reference" for an explanation of the ***__segment*** keyword.

__segname

WATCOM C/386 supports the ***__segname*** keyword which is used when describing segname constant based pointers or objects. See the "WATCOM C Language Reference" for an explanation of the ***__segname*** keyword.

__self

WATCOM C/386 supports the ***__self*** keyword which is used when describing self based pointers. See the "WATCOM C Language Reference" for an explanation of the ***__self*** keyword.

__cdecl

WATCOM C/386 supports the ***__cdecl*** keyword to describe C functions that are called using a special convention described by a pragma in the "stddef.h" header file.

---

7 *Any keywords marked with a "†" must not be used for PenPoint application development.*

WATCOM C/386 predefines the macros ***cdecl*** and ***_cdecl*** to be equivalent to the ***__cdecl*** keyword.

__pascal WATCOM C/386 supports the ***__pascal*** keyword to describe Pascal functions that are called using a special convention described by a pragma in the "stddef.h" header file.

WATCOM C/386 predefines the macros ***pascal*** and ***_pascal*** to be equivalent to the ***__pascal*** keyword.

__fortran † WATCOM C/386 supports the ***__fortran*** keyword to describe FORTRAN functions that are called using a special convention described by a pragma in the "stddef.h" header file.

WATCOM C/386 predefines the macros ***fortran*** and ***_fortran*** to be equivalent to the ***__fortran*** keyword.

__interrupt † WATCOM C/386 supports the ***__interrupt*** keyword to describe a function that is an interrupt handler.

Example:

```
#include <i86.h>

void __interrupt int10( union INTPACK r )
{
        .
        .
        .
}
```

The code generator will emit instructions to save all registers. The registers are saved on the stack in a specific order so that they may be referenced using the "INTPACK" union as shown in the example above. The code generator will emit instructions to establish addressability to the program's data segment since the DS segment register contents are unpredictable. The function will return using an "IRETD" (interrupt return) instruction.

WATCOM C/386 predefines the macros ***interrupt*** and ***_interrupt*** to be equivalent to the ***__interrupt*** keyword.

__export WATCOM C/386 supports the ***__export*** keyword to describe functions and other object names that are to be exported from a Microsoft Windows, OS/2 Dynamic Link Library (DLL), or PenPoint Dynamic Link Library (DLL). See also the description of the "zu" option.

Example:

```
void __export _Setcolor( int color )
{
        .
        .
        .
}
```

WATCOM C/386 predefines the macro ***_export*** to be equivalent to the ***__export*** keyword.

__loadds † WATCOM C/386 supports the ***__loadds*** keyword to describe functions that require specific loading of the DS register to establish addressability to the function's data segment. This keyword is useful in describing a function that will be placed in a Microsoft Windows, OS/2 Dynamic Link Library (DLL), or PenPoint Dynamic Link Library (DLL). See also the description of the "nd" and "zu" options.

Example:

```
void __export __loadds _Setcolor( int color )
{
        .
        .
        .
}
```

If the function in an OS/2 Dynamic Link Library requires access to private data, the data segment register must be loaded with an appropriate value since it will contain the DS value of the calling application upon entry to the function.

WATCOM C/386 predefines the macro ***_loadds*** to be equivalent to the ***__loadds*** keyword.

__saveregs † WATCOM C/386 recognizes the ***__saveregs*** keyword which is an attribute used by other C compilers to describe a function that must save and restore all registers. By default, all registers are saved and restored by routines compiled with WATCOM C/386.

WATCOM C/386 predefines the macro ***_saveregs*** to be equivalent to the ***__saveregs*** keyword.

__syscall † The ***__syscall*** keyword may be used with function definitions, and indicates that the calling convention used is compatible with functions provided by OS/2 2.x. This calling convention may be controlled by a #pragma directive.

__far16 † WATCOM C/386 recognizes the ***__far16*** keyword which is can be used to define far 16-bit pointers (16-bit selector with 16-bit offset) or far 16-bit function prototypes. This keyword can be used under OS/2 2.0 to call 16-bit functions from your 32-bit flat model program. Integer arguments will automatically be converted to 16-bit integers, and 32-bit pointers will be converted to far 16-bit pointers before calling a special thunking layer to transfer control to the 16-bit function.

_Seg16 † WATCOM C/386 recognizes the ***_Seg16*** keyword which has the same function as the ***__far16*** keyword described above.

3.12 The WATCOM Code Generator

The WATCOM C/386 compiler system consists of two components which we call the "front end" and the "back end". The front end is a compiler which understands the syntax and semantics of the ANSI C programming language. The front end translates your specification of a program into an intermediate form which it transmits to the back end. The back end is a code generator which transforms this intermediate form into machine language instructions and emits these instructions to an object file.

When using the OS/2 or DOS protected-mode versions of the compiler, the front end and back end are combined in a single process.

When using the DOS real-mode version of the compiler, this scheme is implemented as two processes which run under DOS. The following diagram illustrates the processes involved in compiling a C source file and producing an object file.

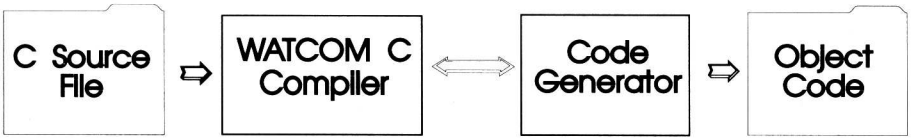


Figure 1. *Code Transformation*

The compiler (first process) starts up the code generator (second process). The compiler, `WCC386.EXE`, will search for, and attempt to load, the code generator `386WCGL.EXE`. The current directory and then all paths listed in the DOS or OS/2 **PATH** environment variable are searched. WATCOM C/386 will print the message

```
Unable to find '386WCGL.EXE'
```

if the code generator cannot be located.

The WATCOM Code Generator has been compiled with the large (big code, big data) memory model. It will make use of all available memory (up to 640K) on the personal computer. The WATCOM Code Generator performs such optimizations as common subexpression elimination, global flow analysis, and so on.

In some cases, the code generator can do a better job of optimizing code if it could utilize more memory. This is indicated when a

```
Not enough memory to optimize procedure 'xxxx'
```

message appears on the screen as the source program is compiled. In such an event, you may wish to make more memory available to the code generator or use the protected-mode version of the compiler (WCC386P).

A special environment variable may be used to obtain memory usage information or set memory usage limits on the code generator. The **WCGMEMORY** environment variable may be used to request a report of the amount of memory used by the compiler's code generator for its work area.

Example:

```
SET WCGMEMORY=?
```

When the memory amount is "?" then the code generator will report how much memory was used to generate the code.

It may also be used to instruct the compiler's code generator to allocate a fixed amount of memory for a work area.

Example:

```
SET WCGMEMORY=128
```

When the memory amount is "nnn" then exactly "nnnK" bytes will be used. In the above example, 128K bytes is requested. If less than "nnnK" is available then the compiler will quit with a fatal error message. If more than "nnnK" is available then only "nnnK" will be used.

There are two reasons why this second feature may be quite useful. In general, the more memory available to the code generator, the more optimal code it will generate. Thus, for two personal computers with different amounts of memory, the code generator may produce different (although correct) object code. If you have a software quality assurance requirement that the same results (i.e., code) be produced on two different machines then you should use this feature. To generate identical code on two personal computers with different memory configurations, you must ensure the following two conditions.

1. Both personal computers must have EMS memory or both computers must *not* have EMS memory.

2. You must set the **WCGMEMORY** environment variable identically on both machines.

The second reason where this feature is useful is on virtual memory paging systems (e.g., OS/2) where an unlimited amount of memory can be used by the code generator. If a very large module is being compiled, it may take a very long time to compile it. The code generator will continue to allocate more and more memory and cause an excessive amount of paging. By restricting the amount of memory that the code generator can use, you can reduce the amount of time required to compile a routine.

3.13 The WATCOM C/386 for PenPoint 1.0 Libraries

The WATCOM C/386 library routines are described in the *WATCOM C Library Reference for PenPoint* manual. The PenPoint kernel functions are described in "PenPoint Architectural Reference, Volume II, Part 9: System Services".

The WATCOM C library for PenPoint is located under the `\PENPOINT\SDK\LIB` directory.

```
          \penpoint\sdk
               |
---+---------+---------+---
   |         |         |
  lib       inc      sample
   |         |         |
```

The name of the library containing the WATCOM C library routines is called `PENPOINT.LIB`.

WATCOM Linker

4 The PenPoint Executable and DLL File Formats

This chapter deals with those aspects of the WATCOM Linker required to generate PenPoint executable files. PenPoint executables use the OS/2 32-bit linear executable format (LE format only). In this chapter, we describe how to generate this executable format.

See also the "PenPoint Application Writing Guide" for information on linking PenPoint applications.

4.1 The WATCOM Linker Command Line

Input to the WATCOM Linker is specified on the command line. The following notation is used to describe the syntax of WATCOM Linker commands.

ABC	All items in upper case are required.
[abc]	The item *abc* is optional.
{abc}	The item *abc* may be repeated zero or more times.
{abc}+	The item *abc* may be repeated one or more times.
a|b|c	One of *a*, *b* or *c* may be specified.
a ::= b	The item *a* is defined in terms of *b*.

The WATCOM Linker command line format is as follows.

```
WLINK {directive}
```

where ***directive*** is any of the following:

```
FILE obj_spec{,obj_spec}
            obj_spec ::= obj_file[(obj_module)] | library_file[(obj_module)]
| LIBFILE obj_file{,obj_file}
| LIBPATH path_name{;path_name}
| LIBRARY library_file{,library_file}
| NAME exe_file
| OPTION option{,option}
            option ::= MAP[=map_file] | NODEFAULTLIBS | PACKCODE=n
                    | NAMELEN=n | VERBOSE | STACK=n | CASEEXACT
                    | UNDEFSOK | DOSSEG | SYMFILE[=symbol_file]
                    | ALIGNMENT=n | HEAPSIZE=n | STUB=stub_name
                    | MODNAME=module_name | OLDLIBRARY=dll_name
                    | ONEAUTODATA | MANYAUTODATA | NOAUTODATA
                    | DESCRIPTION 'description' | NEWFILES
                    | OFFSET | PROTMODE | VERSION=major[.minor]
| FORMAT OS2 [exe_type] [dll_form | exe_attrs]
            exe_type ::= FLAT | LE | LX
            dll_form ::= DLL [INITGLOBAL | INITINSTANCE]
                        [TERMINSTANCE | TERMGLOBAL]
            exe_attrs ::= PM | PMCOMPATIBLE | FULLSCREEN
                        | PHYSDEVICE | VIRTDEVICE
| NEWSEGMENT
| SEGMENT seg_desc{,seg_desc}
            seg_desc ::= seg_id {seg_attrs}+
            seg_id ::= 'seg_name' | CLASS 'class_name'
            seg_attrs ::= PRELOAD | LOADONCALL
                        | IOPL | NOIOPL
                        | EXECUTEONLY | EXECUTEREAD
                        | READONLY | READWRITE
                        | SHARE | NOSHARE
                        | CONFORMING | NONCONFORMING
| IMPORT import{,import}
            import ::= internal_name module_name[.entry_name | n]
| EXPORT export{,export}
            export ::= entry_name[.n][=internal_name] [RESIDENT] [n]
| MODTRACE obj_module{,obj_module}
| SYMTRACE symbol_name{,symbol_name}
| SYSTEM BEGIN system_name {directive} END
| SYSTEM system_name
| PATH path_name{;path_name}
| DEBUG [db_option{,db_option}]
            db_option ::= LINES | TYPES | LOCALS | ALL
| @ directive_file
| # comment
```

where:	*description:*
class_name	is a class name.
comment	is any sequence of characters.
description	is a sequence of characters.
directive_file	is a file specification for the name of a linker directive file. If no file extension is specified, a file extension of "lnk" is assumed.
dll_name	is a file specification for the name of a dynamic link library. If no file extension is specified, a file extension of "dll" is assumed.
entry_name	is a function name.
exe_file	is a file specification for the name of the executable file. If no file extension is specified, a file extension of "exe" is assumed. If a dynamic link library file is being generated, a file extension of "dll" is assumed.
internal_name	is a function name.
library_file	is a file specification for the name of a library file. If *library_file* appears in a "LIBRARY" directive and no file extension is specified, a file extension of "lib" is assumed. If *library_file* appears in a "FILE" directive and no file extension is specified, a file extension of "obj" is assumed if you are running a DOS or OS/2-hosted version of the WATCOM Linker. A file extension of "o" is assumed if you are running a QNX-hosted version of the WATCOM Linker. When a library file is specified in a "FILE" directive and *obj_module* is specified, the object module identified by *obj_module* is extracted from the library file and included in the executable file. If *obj_module* is not specified (only the library file is specified), all object modules in the library are included in the executable file.
major	specifies the major version number.
map_file	is a file specification for the name of the map file. If no file extension is specified, a file extension of "map" is assumed.
minor	specifies the minor version number and must be less than 100.
module_name	is the name of a dynamic link library. Note that this need not be the same as the file name of the executable file that contains the dynamic link library.

n represents a value. The complete form of *n* is the following.

```
[0x]d{d}[k]
```

d represents a decimal digit. If *0x* is specified, the string of digits represents a hexadecimal number. If *k* is specified, the value is multiplied by 1024.

obj_file is a file specification for the name of an object file. If no file extension is specified, a file extension of "obj" is assumed if you are running a DOS or OS/2-hosted version of the WATCOM Linker. A file extension of "o" is assumed if you are running a QNX-hosted version of the WATCOM Linker.

obj_module is the name of an object module contained in a library file or object file.

Object files may contain multiple object modules. A simple way of creating such an object file is to concatenate a number of object files into a single object file. Each of the original object files is now an object module in the resulting object file. Also, some language processors may generate object files that contain multiple object modules. Specifying *obj_module* allows you to select a particular object module from an object file.

path_name is a path name.

seg_name is the name of the code or data segment whose attributes are being specified.

stub_name is a file specification for the name of the executable file. If no file extension is specified, a file extension of "exe" is assumed.

symbol_file is a file specification for the name of the symbol file. If no file extension is specified, a file extension of "sym" is assumed.

symbol_name is the name of a symbol.

system_name is the name of a system.

You can view all the directives specific to PenPoint executable files by simply typing the following if you are using a DOS or OS/2-hosted version of the WATCOM Linker

```
wlink ? os2
```

and typing the following if you are using a QNX-hosted version of the WATCOM Linker.

```
use wlink os2
```

Notes:

1. If the file "wlink.hlp" is located in one of the paths specified in the "PATH" environment variable, the contents of that file will be displayed when the following command is issued.

   ```
   wlink ?
   ```

2. If all of the directive information does not fit on the command line, type the following.

   ```
   wlink
   ```

   The prompt "WLINK>" will appear on the next line. You can enter as many lines of directive information as required. Press "Ctrl/Z" followed by the "Enter" key to terminate the input of directive information if you are running a DOS or OS/2-hosted version of the WATCOM Linker. Press "Ctrl/D" to terminate the input of directive information if you are running a QNX-hosted version of the WATCOM Linker.

4.2 Dynamic Link Libraries

The WATCOM Linker can generate two forms of executable files; program modules and dynamic link libraries. A program module is the executable file that gets loaded by PenPoint when you run your application. A dynamic link library is really a library of routines that are called by a program module but not linked into the program module. The executable code in a dynamic link library gets loaded by PenPoint during the execution of a program module when a routine in the dynamic link library is called.

Program modules are contained in files whose name has a file extension of "exe". Dynamic link libraries are contained in files whose name has a file extension of "dll". The WATCOM Linker "FORMAT" directive can be used to select the type of executable file to be generated.

Let us consider some of the advantages of using dynamic link libraries over standard libraries.

1. Functions in dynamic link libraries are not linked into your program. Only references to the functions in dynamic link libraries are placed in the program module. These references are called import definitions. As a result, the linking time is reduced and disk space is saved. If many applications reference the same dynamic link library, the saving in disk space can be significant.

2. Since program modules only reference dynamic link libraries and do not contain the actual executable code, a dynamic link library can be updated without re-linking your application. When your application is executed, it will use the updated version of the dynamic link library.

3. Dynamic link libraries also allow sharing of code and data between the applications that use them. If many applications that use the same dynamic link library are executing concurrently, the sharing of code and data segments improves memory utilization.

4.2.1 Creating Dynamic Link Libraries

To create a dynamic link library, you must specify the following form of the "FORMAT" directive.

```
format os2 dll
```

In addition, you must specify which functions in the dynamic link library are to be made available to applications which use it. This is achieved by using the "EXPORT" directive for each function that can be called by an application.

Dynamic link libraries can reference other dynamic link libraries. References to other dynamic link libraries are resolved by specifying "IMPORT" directives or using import libraries.

4.2.2 Using Dynamic Link Libraries

To use a dynamic link library, you must tell the WATCOM Linker which functions are contained in a dynamic link library and the name of the dynamic link library. This is achieved in two ways.

The first method is to use the "IMPORT" directive. The "IMPORT" directive names the function and the dynamic link library it belongs to so that the WATCOM Linker can generate an import definition in the program module.

The second method is to use import libraries. An import library is a standard library which contains object modules with special object records that define the functions belonging to a dynamic link library. An import library is created from a dynamic link library using the WATCOM Library Manager. The resulting import library can then be specified in a "LIBRARY" directive in the same way one would specify a standard library. See the chapter entitled "The library Manager" for more information on creating import libraries.

Using an import library is the preferred method of providing references to functions in dynamic link libraries. When a dynamic link library is modified, typically the import library corresponding to the modified dynamic link library is updated to reflect the changes. Hence, any directive file that specifies the import library in a "LIBRARY" directive need not be modified. However, if you are using "IMPORT" directives, you may have to modify the "IMPORT" directives to reflect the changes in the dynamic link library.

4.3 WATCOM Linker Directives

Directives tell the WATCOM Linker how to create your program. For example, using directives, you can tell the WATCOM Linker which object files are to be included in the program, which library files to search to resolve undefined references, and the name of the executable file.

The file "wlink.lnk" is a special linker directive file that is automatically processed by the WATCOM Linker before processing any other directives. This file must be located in one of the paths specified in the "PATH" environment variable so that the WATCOM Linker can locate it. A default "wlink.lnk" is located in the "binb" directory of the directory in which you installed your software.

Note that "wlink.lnk" references the "WATCOM" environment variable which must be set to the directory in which you installed your software.

It is also possible to use environment variables when specifying a directive. For example, if the "LIBDIR" environment variable is defined as follows,

```
set libdir=\test
```

then the linker directive

```
library %libdir%\mylib
```

is equivalent to the following linker directive.

```
library \test\mylib
```

Note that a space must precede a reference to an environment variable.

The following sections describe those WATCOM Linker directives that are used to generate PenPoint executable files.

4.3.1 The DEBUG Directive

The "DEBUG" directive is used to tell the WATCOM Linker to generate debugging information in the executable file. This extra information in the executable file is used by the WATCOM debugger, VIDEO. The format of the "DEBUG" directive (short form "D") is as follows.

DEBUG [db_option{,db_option}]

db_option ::= LINES | TYPES | LOCALS | ALL

The following lists the classes of debugging information.

- global symbol information
- line numbering information
- local symbol information
- typing information

The following options can be used with the "DEBUG" directive to control which of the above classes of debugging information is included in the executable file.

LINES (short form "LI") specifies line numbering and global symbol information.

LOCALS (short form "LO") specifies local and global symbol information.

TYPES (short form "T") specifies typing and global symbol information.

ALL (short form "A") specifies all of the above debugging information.

Note: The position of the "DEBUG" directive is important. The level of debugging information specified in a "DEBUG" directive only applies to object files and libraries that appear in *subsequent* "FILE" or "LIBRARY" directives. For example, if "DEBUG ALL" was the only "DEBUG" directive specified and was also the last linker directive, no debugging information would appear in the executable file.

Only global symbol information is actually produced by the WATCOM Linker; the other three classes of debugging information are extracted from object modules and copied to the executable file. Therefore, at compile time, you must instruct the compiler to generate local symbol, line numbering and typing information in the object file so that the information can be transferred to the executable file. If you have asked the WATCOM Linker to produce a particular class of debugging information and it appears that none is present, one of the following conditions may exist.

1. The debugging information is not present in the object files.
2. The "DEBUG" directive has been misplaced.

The following sections describe the classes of debugging information.

4.3.1.1 Line Numbering Information - DEBUG LINES

The "DEBUG LINES" option controls the processing of line numbering information. Line numbering information is the line number and address of the generated code for each line of source code in a particular module. This allows VIDEO to perform source-level debugging. When the WATCOM Linker encounters a "DEBUG" directive with a "LINES" or "ALL" option, line number information for each subsequent object module will be placed in the executable file. This includes all object modules extracted from object files specified in subsequent "FILE" directives and object modules extracted from libraries specified in subsequent "LIBRARY" or "FILE" directives.

> ***Note:*** All modules for which line numbering information is requested must have been compiled with the "d1" or "d2" option.

A subsequent "DEBUG" directive without a "LINES" or "ALL" option terminates the processing of line numbering information.

4.3.1.2 Local Symbol Information - DEBUG LOCALS

The "DEBUG LOCALS" option controls the processing of local symbol information. Local symbol information is the name and address of all symbols local to a particular module. This allows VIDEO to locate these symbols so that you can reference local data and routines by name. When the WATCOM Linker encounters a "DEBUG" directive with a "LOCALS" or "ALL" option, local symbol information for each subsequent object module will be placed in the executable file. This includes all object modules extracted from object files specified in subsequent "FILE" directives and object modules extracted from libraries specified in subsequent "LIBRARY" or "FILE" directives.

> ***Note:*** All modules for which local symbol information is requested must have been compiled with the "d2" option.

A subsequent "DEBUG" directive without a "LOCALS" or "ALL" option terminates the processing of local symbol information.

4.3.1.3 Typing Information - DEBUG TYPES

The "DEBUG TYPES" option controls the processing of typing information. Typing information includes a description of all types, structures and arrays that are defined in a module. This allows VIDEO to display variables according to their type. When the WATCOM Linker encounters a "DEBUG" directive with a "TYPES" or "ALL" option, typing information for each subsequent object module will be placed in the executable file. This includes all object modules extracted from object files specified in subsequent "FILE" directives and object modules extracted from libraries specified in subsequent "LIBRARY" or "FILE" directives.

> ***Note:*** All modules for which typing information is requested must have been compiled with the "d2" option.

A subsequent "DEBUG" directive without a "TYPES" or "ALL" option terminates the processing of typing information.

4.3.1.4 All Debugging Information - DEBUG ALL

The "DEBUG ALL" option specifies that "LINES", "LOCALS" and "TYPES" options are requested. The "LINES" option controls the processing of line numbering information. The "LOCALS" option controls the processing of local symbol information. The "TYPES" option controls the processing of typing information. Each of these options is described in a previous section. A subsequent "DEBUG" directive without an "ALL" option discontinues those options which are not specified in the list of debug options.

4.3.1.5 Global Symbol Information

Global symbol information consists of all the global symbols in your program and their address. This allows VIDEO to locate these symbols so that you can reference global data and routines by name. When the WATCOM Linker encounters a "DEBUG" directive, global symbol information for all the global symbols appearing in your program is placed in the executable file.

4.3.1.6 Using DEBUG Directives

Consider the following directive file.

```
debug all
file module1
debug lines
file module2, module3
debug
library mylib
```

It specifies that the following debugging information is to be generated in the executable file.

1. global symbol information for your program

2. line numbering, typing and local symbol information for the following object files:

   ```
   module1.obj
   ```

3. line numbering information for the following object files:

   ```
   module2.obj
   module3.obj
   ```

Note that if the "DEBUG" directive before the "LIBRARY" directive is not specified, line numbering information for all object modules extracted from the library "mylib.lib" would be generated in the executable file provided the object modules extracted from the library have line numbering information present.

> ***Note:*** A "DEBUG" directive with no option suppresses the processing of line numbering, local symbol and typing information for all subsequent object modules.

Debugging information can use a significant amount of disk space. As shown in the above example, you can select only the class of debugging information you want and for those modules you wish to debug. In this way, the amount of debugging information in the executable file is minimized and hence the amount of disk space used by the executable file is kept to a minimum.

As you can see from the above example, the position of the "DEBUG" directive is important when describing the debugging information that is to appear in the executable file.

> ***Note:*** If you want all classes of debugging information for all files to appear in the executable file you must specify "DEBUG ALL" before any "FILE" and "LIBRARY" directives.

4.3.1.7 Removing Debugging Information from an Executable File

A utility called WSTRIP has been provided which takes as input an executable file and removes the debugging information placed in the executable file by the WATCOM Linker.

For more information on this utility, see the chapter entitled "The Strip Utility".

4.3.2 The EXPORT Directive

The "EXPORT" directive can be used to define the names and attributes of functions in dynamic link libraries that are to be exported. An "EXPORT" definition must be specified for every dynamic link library function that is to be made available externally.

The format of the "EXPORT" directive (short form "EXP") is as follows.

> ***EXPORT export{,export}***
>
> ***export ::= entry_name[.ordinal][=internal_name] [RESIDENT] [iopl_words]***

where:	*description:*
entry_name	is the name to be used by other applications to call the function.
ordinal	is the ordinal value of the function. If the ordinal number is specified, other applications can reference the function by using this ordinal number.
internal_name	is the actual name of the function and should only be specified if it differs from the entry name.
RESIDENT	specifies that the function's entry name should be kept resident in memory. This applies only if the ordinal is specified. If no ordinal is specified, the entry name is always memory resident. Memory resident entry names allow PenPoint to resolve calls more efficiently when the call is by entry name rather than by ordinal.
iopl_words	is required for functions that execute with I/O privilege. *iopl_words* specifies that total size of the function's arguments in words. When such a function is executed, the specified number of words is copied from the caller's stack to the I/O-privileged function's stack. The maximum number of words allowed is 63.

> ***Note:*** By default, the WATCOM C compiler appends an underscore ('_') to all function names. This should be considered when specifying *entry_name* and *internal_name* in an "EXPORT" directive.

4.3.3 The FILE Directive

The "FILE" directive is used to specify the object files and library modules that the WATCOM Linker is to process. The format of the "FILE" directive (short form "F") is as follows.

```
FILE obj_spec{,obj_spec}

obj_spec ::= obj_file[(obj_module)]
           | library_file[(obj_module)]
```

where:	*description:*
obj_file	is a file specification for the name of an object file. If no file extension is specified, a file extension of "obj" is assumed if you are running a DOS or OS/2-hosted version of the WATCOM Linker. A file extension of "o" is assumed if you are running a QNX-hosted version of the WATCOM Linker.
library_file	is a file specification form the name of a library file. Note that the file extension of the library file (usually "lib") must be specified; otherwise an object file will be assumed.
obj_module	is the name of an object module defined in an object or library file.

Consider the following example.

Example:

```
wlink form os2 f \math\sin, mycos
```

The WATCOM Linker is instructed to process the following object files:

```
\math\sin.obj
mycos.obj
```

The object file "mycos.obj" is located in the current directory since no path was specified.

More than one "FILE" directive may be used. The following example is equivalent to the preceding one.

Example:

```
wlink form os2 f \math\sin f mycos
```

Thus, other directives may be placed between lists of object files.

The "FILE" directive can also specify object modules from a library file or object file. Consider the following example.

Example:

```
wlink form os2 f \math\math.lib(sin)
```

The WATCOM Linker is instructed to process the object module "sin" contained in the library file "math.lib" in the directory "\math".

In the following example, the WATCOM Linker will process the object module "sin" contained in the object file "math.obj" in the directory "\math".

Example:

```
wlink form os2 f \math\math(sin)
```

In the following example, the WATCOM Linker will include all object modules contained in the library file "math.lib" in the directory "\math".

Example:

```
wlink form os2 f \math\math.lib
```

4.3.4 The FORMAT Directive

The "FORMAT" directive is used to specify the format of the executable file that the WATCOM Linker is to generate. The format of the "FORMAT" directive (short form "FORM") is as follows.

```
FORMAT form

form ::= DOS [COM]
       | WINDOWS [win_dll] [MEMORY] [FONT]
       | OS2 [exe_type] [os2_dll | exe_attrs]
       | PHARLAP [EXTENDED | REX]
       | NOVELL [NLM | LAN | DSK | NAM] 'description'
       | QNX

os2_dll ::= DLL [INITGLOBAL | INITINSTANCE]
        [TERMINSTANCE | TERMGLOBAL]

win_dll ::= DLL [INITGLOBAL | INITINSTANCE]

exe_type ::= FLAT | LE | LX

exe_attrs ::= PM | PMCOMPATIBLE | FULLSCREEN
            | PHYSDEVICE | VIRTDEVICE
```

where: ***description:***

DOS (short form "D") tells the WATCOM Linker to generate a DOS "EXE" file. For more information on DOS executable file formats, see the chapter entitled "The DOS Executable File Format".

WINDOWS tells the WATCOM Linker to generate a Windows executable file. For more information on Windows executable file formats, see the chapter entitled "The Windows Executable File Format".

OS2 tells the WATCOM Linker to generate an OS/2 executable file format. The name of the executable file will have extension "exe". If "LE" is specified, an early form of the OS/2 32-bit linear executable will be generated. This executable file format is required by PenPoint and Rational's DOS/4G DOS extender.

In order to improve load time and minimize the size of the executable file, the OS/2 32-bit linear executable file format was changed. If "LX" or "FLAT" (short form "FL") is specified, the new form of the OS/2 32-bit linear executable will be generated.

If "FLAT", "LX" or "LE" is not specified, an OS/2 16-bit executable will be generated.

If "DLL" (short form "DL") is specified, a dynamic link library will be generated in which case the name of the executable file will have extension "dll". Note that these default extensions can be overridden by using the "NAME" directive to name the executable file.

Specifying INITGLOBAL (short form "INITG") will cause the initialization routine to be called the first time the dynamic link library is loaded. Specifying INITINSTANCE (short form "INITI") will cause the initialization routine to be called each time the dynamic link library is referenced by a process. If neither "INITGLOBAL" or "INITINSTANCE" is specified, "INITGLOBAL" is assumed. For OS/2 32-bit linear executable files, it is also possible to specify whether the initialization routine is to be called at DLL termination or not. Specifying TERMGLOBAL (short form "TERMG") will cause the initialization routine to be called when the last instance of the dynamic link library is terminated. Specifying TERMINSTANCE (short form "TERMI") will cause the initialization routine to be called each time an instance of the dynamic link library is terminated. Note that the initialization routine is passed an argument indicating whether it is being called during DLL initialization or DLL termination. If "INITINSTANCE" is used and no termination option is specified, "TERMINSTANCE" is assumed. If "INITGLOBAL" is used and no termination option is specified, "TERMGLOBAL" is assumed.

If "PM" is specified, a Presentation Manager application will be created. The application uses the API provided by the Presentation Manager and must be executed in the Presentation Manager environment.

If "PMCOMPATIBLE" (short form "PMC") is specified, an application compatible with Presentation Manager will be created. The application can run inside the Presentation Manager or it can run in a separate screen group. An application can be of this type if it uses the proper subset of OS/2 video, keyboard, and mouse functions supported in the Presentation Manager applications. This is the default.

If "FULLSCREEN" (short form "FULL") is specified, an OS/2 full screen application will be created. The application will run in a separate screen group from the Presentation Manager.

If "PHYSDEVICE" (short form "PHYS") is specified, the executable file is marked as a physical device driver.

If "VIRTDEVICE" (short form "VIRT") is specified, the executable file is marked as a virtual device driver.

PHARLAP (short form "PHAR") tells the WATCOM Linker to generate an executable file that will run under Phar Lap's 386|DOS-Extender. For more information on Phar Lap executable file formats, see the chapter entitled "The Phar Lap Executable File Format".

NOVELL (short form "NOV") tells the WATCOM Linker to generate a NetWare 386 executable file, more commonly called a NetWare Loadable Module (NLM). For more information on NetWare 386 executable file formats, see the chapter entitled "The Novell NetWare 386 Executable File Format".

QNX tells the WATCOM Linker to generate a QNX executable file. For more information on QNX executable file formats, see the chapter entitled "The QNX Executable File Format".

If no "FORMAT" directive is specified and you are running a DOS or OS/2-hosted version of the WATCOM Linker, the executable file format will be selected in the following way.

1. If a reference to a dynamic link library is encountered, an OS/2 executable file will be generated.
2. If a 386 object module is encountered, an executable file that runs under Phar Lap's 386|DOS-Extender will be generated.
3. Otherwise, a DOS executable file will be generated.

If no "FORMAT" directive is specified and you are running a QNX-hosted version of the WATCOM Linker, the executable file format will be selected in the following way.

1. If a reference to a dynamic link library is encountered, an OS/2 format executable file will be generated.
2. If a 386 object module is encountered, an executable file that runs under Phar Lap's 386|DOS-Extender will be generated.
3. Otherwise, a QNX executable file will be generated.

4.3.5 The IMPORT Directive

The "IMPORT" directive describes a function that belongs to a dynamic link library. The format of the "IMPORT" directive (short form "IMP") is as follows.

```
IMPORT import{,import}

import ::= internal_name module_name[.entry_name | ordinal]
```

where: ***description:***

internal_name is the name the application used to call the function.

module_name is the name of the dynamic link library. Note that this need not be the same as the file name of the executable file containing the dynamic link library. This name corresponds to the name specified by the "MODNAME" option when the dynamic link library was created.

entry_name is the actual name of the function as defined in the dynamic link library.

ordinal is the ordinal value of the function. The ordinal number is an alternate method that can be used to reference a function in a dynamic link library.

> *Note:* By default, the WATCOM C compiler appends an underscore ('_') to all function names. This should be considered when specifying *internal_name* and *entry_name* in an "IMPORT" directive.

The preferred method to resolve references to dynamic link libraries is through the use of import libraries. See the section entitled "Using Dynamic Link Libraries" for more information on import libraries.

4.3.6 The LIBRARY Directive

The "LIBRARY" directive is used to specify the library files to be searched when unresolved symbols remain after processing all specified input object files. The format of the "LIBRARY" directive (short form "L") is as follows.

> ***LIBRARY library_file{,library_file}***

where: ***description:***

library_file is a file specification for the name of a library file. If no file extension is specified, a file extension of "lib" is assumed.

Consider the following example.

Example:

```
wlink form os2 f trig l \math\trig, \cmplx\trig
```

The WATCOM Linker is instructed to process the following object file:

```
trig.obj
```

If any unresolved symbol references remain after all object files have been processed, the following library files will be searched:

```
\math\trig.lib
\cmplx\trig.lib
```

More than one "LIBRARY" directive may be used. The following example is equivalent to the preceding one.

Example:

```
wlink form os2 f trig l \math\trig l \cmplx\trig
```

Thus other directives may be placed between lists of library files.

4.3.6.1 Searching for Libraries Specified in Environment Variables

The "LIB" environment variable can be used to specify a list of paths that will be searched for library files. The "LIB" environment variable can be set using the "set" command as follows:

```
set lib=\graphics\lib;\utility
```

Consider the following "LIBRARY" directive and the above definition of the "LIB" environment variable.

```
library \mylibs\util, graph
```

If undefined symbols remain after processing all object files specified in all "FILE" directives, the WATCOM Linker will resolve these references by searching the following libraries in the specified order.

1. the library file "\mylibs\util.lib"
2. the library file "graph.lib" in the current directory
3. the library file "\graphics\lib\graph.lib"
4. the library file "\utility\graph.lib"

Notes:

1. If a library file specified in a "LIBRARY" directive contains an absolute path specification, the WATCOM Linker will not search any of the paths specified in the "LIB" environment string for the library file. An absolute path specification is one that begins with a drive specification or the "\" character.

2. Once a library file has been found, no further elements of the "LIB" environment variable are searched for other libraries of the same name. That is, if the library file "\graphics\lib\graph.lib" exists, the library file "\utility\graph.lib" will not be searched even though unresolved references may remain.

Since the WATCOM Linker can generate various executable file formats, it may be necessary to specify a different set of libraries for each executable file format. For this reason, the directories specified in the following environment variables will be searched.

LIBDOS The directories specified in the "LIBDOS" environment variable will be searched if the WATCOM Linker is generating a DOS executable file.

LIBOS2 The directories specified in the "LIBOS2" environment variable will be searched if the WATCOM Linker is generating an OS/2 executable file.

LIBPHAR The directories specified in the "LIBPHAR" environment variable will be searched if the WATCOM Linker is generating a Phar Lap executable file.

LIBNOV The directories specified in the "LIBNOV" environment variable will be searched if the WATCOM Linker is generating a NetWare 386 executable file.

LIBQNX The directories specified in the "LIBQNX" environment variable will be searched if the WATCOM Linker is generating a QNX executable file.

LIBWIN The directories specified in the "LIBWIN" environment variable will be searched if the WATCOM Linker is generating a Windows executable file.

Note that the directories specified in the above mentioned environment variables will be searched before the directories specified in the "LIB" environment variable.

4.3.7 The LIBFILE Directive

The "LIBFILE" directive is used to specify the object files that the WATCOM Linker is to process. The format of the "LIBFILE" directive (short form "LIBF") is as follows.

LIBFILE obj_file{,obj_file}

where: ***description:***

obj_file is a file specification for the name of an object file. If no file extension is specified, a file extension of "obj" is assumed if you are running a DOS or OS/2-hosted version of the WATCOM Linker. A file extension of "o" is assumed if you are running a QNX-hosted version of the WATCOM Linker.

The difference between the "LIBFILE" directive and the "FILE" directive is as follows.

1. The "LIBFILE" directive only allows object files. The "FILE" directive allows object files as well as a syntax for specifying object modules from library files and object files.

2. When searching for an object file specified in a "LIBFILE" directive, the default directory will be searched first, followed by the paths specified in the "LIBPATH" directive, and finally the paths specified in the appropriate "LIB" environment variable. Note that if the object file name contains a path, only the specified path will be searched.

3. Object files specified in a "LIBFILE" directive will not be used to create the name of the executable file when no "NAME" directive is specified.

Essentially, object files that appear in "LIBFILE" directives are are viewed as components of a library that have not been explicitly placed in a library file.

Consider the following linker directive file.

```
libpath \libs
libfile mystart
path \objs
file file1, file2
```

The WATCOM Linker is instructed to process the following object files:

```
\libs\mystart.obj
\objs\file1.obj
\objs\file2.obj
```

Note that the executable file will have file name "file1" and not "mystart".

4.3.8 The LIBPATH Directive

The "LIBPATH" directive is used to specify the directories that are to be searched for library files appearing in subsequent "LIBRARY" directives and object files appearing in subsequent "LIBFILE" directives. The format of the "LIBPATH" directive (short from "LIBP") is as follows.

LIBPATH [path_name{;path_name}]

where: ***description:***

path_name is a path name.

Consider a directive file containing the following linker directives.

```
file test
libpath \math
library trig
libfile newsin
```

First, the WATCOM Linker will process the object file "test.obj" from the default directory. The object file "newsin.obj" will then be processed, searching the default directory first. If "newsin.obj" is not in the default directory, the "\math" directory will be searched. If any unresolved references remain after processing the object files, the library file "trig.lib" will be searched. If the file "trig.lib" does not exist in the default directory, the "\math" directory will be searched.

It is also possible to specify a list of paths in a "LIBPATH" directive. Consider the following example.

```
libpath \newmath;\math
library trig
```

When processing undefined references, the WATCOM Linker will attempt to process the library file "trig.lib" in the default directory. If "trig.lib" does not exist in the default

directory, the "\newmath" directory will be searched. If "trig.lib" does not exist in the "\newmath" directory, the "\math" directory will be searched.

If the name of a library file appearing in a "LIBRARY" directive or the the name of an object file appearing in a "LIBFILE" directive contains a path specification, only the specified path will be searched.

Note that

```
libpath path1
libpath path2
```

is equivalent to the following.

```
libpath path2;path1
```

4.3.9 The MODTRACE Directive

The "MODTRACE" directive instructs the WATCOM Linker to print a list of all modules that reference the symbols defined in the specified modules. The format of the "MODTRACE" directive (short form "MODT") is as follows.

MODTRACE module_name{,module_name}

where: ***description:***

module_name is the name of an object module defined in an object or library file.

The information is displayed in the map file. Consider the following example.

Example:

```
wlink form os2 op map f test l math modt trig
```

If the module "trig" defines the symbols "sin" and "cos", the WATCOM Linker will list, in the map file, all modules that reference the symbols "sin" and "cos".

Note:

1. The "MODTRACE" directive cannot be used in conjunction with the "VERBOSE" option.

4.3.10 The NAME Directive

The "NAME" directive is used to provide a name for the executable file generated by the WATCOM Linker. The format of the "NAME" directive (short form "N") is as follows.

NAME exe_file

where:	***description:***
exe_file	is a file specification for the name of the executable file. If no file extension is specified, a file extension of "exe" is assumed unless the "FORMAT" directive is used to change the executable file format.

Consider the following example.

Example:

```
wlink form os2 n myprog f test, test2, test3
```

The WATCOM Linker is instructed to generate an executable file called "myprog.exe".

Notes:

1. No file extension was given when the executable file name was specified. The WATCOM Linker assumes a file extension that depends on the format of the executable file being generated. The section entitled "The FORMAT Directive" describes the "FORMAT" directive and how the file extension is chosen for each executable file format.

2. If no "NAME" directive is present, the executable file will have the file name of the first object file processed by the WATCOM Linker. If the first object file processed is called "test.obj" and no "NAME" directive is specified, an executable file called "test.exe" will be generated.

4.3.11 The NEWSEGMENT Directive

By default, the WATCOM Linker automatically groups logical code segments into physical segments. The "PACKCODE" option can be used to specify the size of the physical segments. Note that all physical segments are the same size.

The "NEWSEGMENT" directive provides an alternate method of grouping code segments into physical segments. By placing this directive after a sequence of "FILE" directives, all code segments appearing in object modules specified by the sequence of "FILE" directives will be packed into a physical segment. Note that the size of a physical segment may vary in size. The format of the "NEWSEGMENT" directive (short form "NEW") is as follows.

NEWSEGMENT

Consider the following example.

```
file file1, file2, file3
newsegment
file file4
file file5
```

Code segments from *file1*, *file2* and *file3* will be grouped into one physical segment. Code segments from *file4* and *file5* will be grouped into another physical segment.

Note that code segments extracted from library files will be grouped into physical segments as well. The size of these physical segments is determined by the "PACKCODE" option and is 64k by default.

4.3.12 The OPTION Directive

The "OPTION" directive is used to specify options to the WATCOM Linker. The format of the "OPTION" directive (short form "OP") is as follows.

```
OPTION option{,option}

option ::= MAP[=map_file] | NODEFAULTLIBS | PACKCODE=n
           | NAMELEN=n | VERBOSE | STACK=n | CASEEXACT
           | UNDEFSOK | DOSSEG | SYMFILE[=symbol_file]
           | ALIGNMENT=n | HEAPSIZE=n | STUB=stub_name
           | MODNAME=module_name | OLDLIBRARY=dll_name
           | ONEAUTODATA | MANYAUTODATA | NOAUTODATA
           | DESCRIPTION 'description'
           | NEWFILES | OFFSET | PROTMODE
           | VERSION=major[.minor]
```

where: ***description:***

map_file is a file specification for the name of the map file. If no file extension is specified, a file extension of "map" is assumed.

symbol_file is a file specification for the name of the symbol file. If no file extension is specified, a file extension of "sym" is assumed.

module_name is the name of a program module or dynamic link library.

dll_name is a file specification for the name of a dynamic link library. If no file extension is specified, a file extension of "dll" is assumed.

stub_name is a file specification for the name of the executable file. If no file extension is specified, a file extension of "exe" is assumed.

description is a sequence of characters.

n represents a value. The complete form of *n* is the following.

```
[0x]d{d}[k]
```

d represents a decimal digit. If *0x* is specified, the string of digits represents a hexadecimal number. If *k* is specified, the value is multiplied by 1024.

The following sections describe the WATCOM Linker options.

4.3.12.1 The ALIGNMENT Option

The "ALIGNMENT" option specifies the alignment for segments in the executable file. The format of the "ALIGNMENT" option (short form "A") is as follows.

OPTION ALIGNMENT=n

where:	***description:***
n	represents a value. The complete form of *n* is the following. `[0x]d{d}[k]` *d* represents a decimal digit. If *0x* is specified, the string of digits represents a hexadecimal number. If *k* is specified, the value is multiplied by 1024.

n specifies the alignment for segments in the executable file and must be a power of 2.

Segments in the executable file are pointed to by a segment table. An entry in the segment table contains a 16-bit value which is a multiple of the alignment value. Together they form the offset of the segment from the start of the segment table. Note that the smaller the value of *n* the smaller the executable file.

By default, the WATCOM Linker will automatically choose the smallest value of *n* possible. You need not specify this option unless you want padding between segments in the executable file.

4.3.12.2 The CASEEXACT Option

The "CASEEXACT" option tells the WATCOM Linker to respect case when resolving references to global symbols. That is, "ScanName" and "SCANNAME" represent two different symbols. By default, the WATCOM Linker is case insensitive; "ScanName" and "SCANNAME" represent the same symbol. The format of the "CASEEXACT" option (short form "C") is as follows.

OPTION CASEEXACT

4.3.12.3 The DESCRIPTION Option

The "DESCRIPTION" option inserts the specified text into the application or dynamic link library. This is useful if you wish to embed copyright information into an application or dynamic link library. The format of the "DESCRIPTION" option (short form "DE") is as follows.

OPTION DESCRIPTION 'description'

where: ***description:***

description is the sequence of characters to be embedded into the application or dynamic link library.

4.3.12.4 The DOSSEG Option

The "DOSSEG" option tells the WATCOM Linker to order segments in a special way. The format of the "DOSSEG" option (short form "D") is as follows.

OPTION DOSSEG

When the "DOSSEG" option is specified, segments will be ordered in the following way.

1. all segments not belonging to group "DGROUP" with class "CODE"
2. all other segments not belonging to group "DGROUP"
3. all segments belonging to group "DGROUP" with class "BEGDATA"
4. all segments belonging to group "DGROUP" not with class "BEGDATA", "BSS" or "STACK"
5. all segments belonging to group "DGROUP" with class "BSS"
6. all segments belonging to group "DGROUP" with class "STACK"

A special segment belonging to class "BEGDATA" is defined when linking with WATCOM run-time libraries. This segment is initialized with the hexadecimal byte pattern "01" and is the first segment in group "DGROUP" so that storing data at location 0 can be detected.

Segments belonging to class "BSS" contain uninitialized data. Note that this only includes uninitialized data in segments belonging to group "DGROUP". Segments belonging to class "STACK" are used to define the size of the stack used for your application. Segments belonging to the classes "BSS" and "STACK" are last in the segment ordering so that uninitialized data need not take space in the executable file.

When using WATCOM run-time libraries, it is not necessary to specify the "DOSSEG" option. One of the object files in the WATCOM run-time libraries contains a special record that specifies the "DOSSEG" option.

If no "DOSSEG" option is specified, segments are ordered in the order they are encountered by the WATCOM Linker.

When the "DOSSEG" option is specified, the WATCOM Linker defines two special variables. `_edata` defines the start of the "BSS" class of segments and `_end` defines the end of the "BSS" class of segments. Your program must not redefine these symbols.

4.3.12.5 The HEAPSIZE Option

The "HEAPSIZE" option specifies the size of the heap required by the application. The format of the "HEAPSIZE" option (short form "H") is as follows.

OPTION HEAPSIZE=n

where:	***description:***
n	represents a value. The complete form of *n* is the following. `[0x]d{d}[k]` *d* represents a decimal digit. If *0x* is specified, the string of digits represents a hexadecimal number. If *k* is specified, the value is multiplied by 1024.

n specifies the size of the heap. The default heap size is 0 bytes. The maximum value of *n* is 65536, the maximum size of a physical segment. Actually, for a particular application, the maximum value of *n* is 65536 less the size of group "DGROUP".

4.3.12.6 The MANYAUTODATA Option

The "MANYAUTODATA" option specifies that a copy of the automatic data segment (default data segment defined by the group "DGROUP"), for the program module or dynamic link library being created, is made for each instance. The format of the "MANYAUTODATA" option (short form "MANY") is as follows.

OPTION MANYAUTODATA

The default for a program module is "MANYAUTODATA" and for a dynamic link library is "ONEAUTODATA".

4.3.12.7 The MAP Option

The "MAP" option controls the generation of a map file. The format of the "MAP" option (short form "M") is as follows.

OPTION MAP[=map_file]

where:	***description:***
map_file	is a file specification for the name of the map file. If no file extension is specified, a file extension of "map" is assumed.

By default, no map file is generated. Specifying this option causes the WATCOM Linker to generate a map file. The map file is simply a memory map of your program. That is, it specifies the relative location of all global symbols in your program. The map file also contains the size of your program.

If no file name is specified, the map file will have a default file extension of "map" and the same file name as the executable file. Note that the map file will be created in the current directory even if the executable file name specified in the "NAME" directive contains a path specification.

Alternatively, a file name can be specified. The following directive instructs the WATCOM Linker to generate a map file and call it "myprog.map" regardless of the name of the executable file.

```
option map=myprog
```

You can also specify a path and/or file extension when using the "MAP=" form of the "MAP" option.

4.3.12.8 The MODNAME Option

The "MODNAME" option specifies a name to be given to the module being created. The format of the "MODNAME" option (short form "MODN") is as follows.

OPTION MODNAME=module_name

where: ***description:***

module_name is the name of a dynamic link library.

Once a module has been loaded (whether it be a program module or a dynamic link library), *mod_name* is the name of the module known to PenPoint. If the "MODNAME" option is not used to specify a module name, the default module name is the name of the executable file without the file extension.

4.3.12.9 The NAMELEN Option

The "NAMELEN" option tells the WATCOM Linker that all symbols must be uniquely identified in the number of characters specified or less. If any symbol fails to satisfy this condition, a warning message will be issued. The warning message will state that a symbol has been defined more than once.

The format of the "NAMELEN" option (short form "NAMEL") is as follows.

OPTION NAMELEN=n

where: ***description:***

n represents a value. The complete form of *n* is the following.

```
[0x]d{d}[k]
```

d represents a decimal digit. If *0x* is specified, the string of digits represents a hexadecimal number. If *k* is specified, the value is multiplied by 1024.

Some computer systems, for example, require that all global symbols be uniquely identified in 8 characters. By specifying an appropriate value for the "NAMELEN" option, you can ease the task of porting your application to other computer systems.

4.3.12.10 The NEWFILES Option

The "NEWFILES" option specifies that the application uses the high-performance file system. The format of the "NEWFILES" option (short form "NEWF") is as follows.

OPTION NEWFILES

4.3.12.11 The NOAUTODATA Option

The "NOAUTODATA" option specifies that no automatic data segment (default data segment defined by the group "DGROUP"), exists for the program module or dynamic link library being created. The format of the "NOAUTODATA" option (short form "NOA") is as follows.

OPTION NOAUTODATA

4.3.12.12 The NODEFAULTLIBS Option

Special object module records that specify default libraries are placed in object files generated by WATCOM compilers. These libraries reflect the memory and floating point model that a source file was compiled for and are automatically searched by the WATCOM Linker when unresolved symbols are detected. These libraries are assumed to exist in the current directory or in one of the paths specified in the "LIBOS2" and "LIB" environment variables. By simply including the directories containing the libraries required to link your application in one of these environment variables, you need not specify any libraries in your linker directive files.

Note that all library files that appear in a "LIBRARY" directive are searched before default libraries. The "NODEFAULTLIBS" option instructs the WATCOM Linker to ignore default libraries. That is, only libraries appearing in a "LIBRARY" directive are searched.

The format of the "NODefaultlibs" option (short form "NOD") is as follows.

OPTION NODEFAULTLIBS

4.3.12.13 The OFFSET Option

This option is allowed only when generating an OS/2 32-bit linear executable.

The "OFFSET" option specifies the preferred base linear address at which the executable will be loaded. The WATCOM Linker will relocate the application for the specified base linear address so that when it is loaded by the operating system, no relocation will be required. This decreases the load time of the application.

If the operating system is unable to load the application at the specified base linear address, it will load it at a different location which will increase the load time since a relocation phase must be performed.

The format of the "OFFSET" option (short form "OFF") is as follows.

OPTION OFFSET=n

where:	*description:*
n	represents a value. The complete form of *n* is the following. `[0x]d{d}[k]` *d* represents a decimal digit. If *0x* is specified, the string of digits represents a hexadecimal number. If *k* is specified, the value is multiplied by 1024.

n specifies the base linear address (in bytes) at which the program is loaded and must be a multiple of 64K. The WATCOM Linker will round the value up to a multiple of 64K if it is not already a multiple of 64K. The default base linear address is 64K.

4.3.12.14 The OLDLIBRARY Option

The "OLDLIBRARY" option is used to preserve the export ordinals for successive versions of a dynamic link library. This ensures that any application that references functions in a dynamic link library by ordinal will continue to execute correctly. The format of the "OLDLIBRARY" option (short form "OLD") is as follows.

OPTION OLDLIBRARY=dll_name

where:	*description:*
dll_name	is a file specification for the name of a dynamic link library. If no file extension is specified, a file extension of "dll" is assumed.

Only the current directory or a specified directory will be searched for dynamic link libraries specified in the "OLDLIBRARY" option.

4.3.12.15 The ONEAUTODATA Option

The "ONEAUTODATA" option specifies that the automatic data segment (default data segment defined by the group "DGROUP") for the program module or dynamic link library being created will be shared by all instances. The format of the "ONEAUTODATA" option (short from "ONE") is as follows.

OPTION ONEAUTODATA

The default for a dynamic link library is "ONEAUTODATA" and for a program module is "MANYAUTODATA".

4.3.12.16 The PACKCODE Option

By default, the WATCOM Linker automatically groups logical code segments into physical segments. The "PACKCODE" option is used to specify the size of the physical segment. The format of the "PACKCODE" option (short form "PAC") is as follows.

OPTION PACKCODE=n

where:	***description:***
n	represents a value. The complete form of *n* is the following. `[0x]d{d}[k]` *d* represents a decimal digit. If *0x* is specified, the string of digits represents a hexadecimal number. If *k* is specified, the value is multiplied by 1024.

n specifies the size of the physical segments into which code segments are packed. The default value of *n* is 64k. Note that this is also the maximum size of a physical segment. To suppress automatic grouping of code segments, specify a value of 0 for *n*.

Notes:

1. Only adjacent segments are packed into a physical segment.
2. Segments belonging to the same group are packed in a physical segment. Segments belonging to different groups are not packed into a physical segment.
3. Segments with different attributes are not packed together unless they are explicitly grouped.

4.3.12.17 The PROTMODE Option

The "PROTMODE" option specifies that the application will only run in protected mode. The format of the "PROTMODE" option (short form "PROT") is as follows.

OPTION PROTMODE

4.3.12.18 The QUIET Option

The "QUIET" option tells the WATCOM Linker to suppress all informational messages. Only warning, error and fatal messages will be issued. By default, the WATCOM Linker issues informational messages. The format of the "QUIET" option (short form "Q") is as follows.

OPTION QUIET

4.3.12.19 The STACK Option

The "STACK" option can be used to increase the size of the stack. The format of the "STACK" option (short form "ST") is as follows.

OPTION STACK=n

where:	***description:***
n	represents a value. The complete form of *n* is the following. `[0x]d{d}[k]` *d* represents a decimal digit. If *0x* is specified, the string of digits represents a hexadecimal number. If *k* is specified, the value is multiplied by 1024.

The default stack size is 4096 bytes for protected-mode 32-bit applications. Otherwise the default stack size is 2048 bytes. During execution of your program, you may get an error message indicating your stack has overflowed. If you encounter such an error, you must link your application again, this time specifying a larger stack size using the "STACK" option.

Example:

```
option stack=8192
```

4.3.12.20 The STUB Option

The "STUB" option specifies an executable file that is to be placed at the beginning of the PenPoint executable file being generated. This program will be executed if the PenPoint module is executed under DOS. The format of the "STUB" option is as follows.

OPTION STUB=stub_name

where:	*description:*
stub_name	is a file specification for the name of the stub executable file. If no file extension is specified, a file extension of "exe" is assumed.

The WATCOM Linker will search all paths specified in the "path" environment variable for the stub executable file. The stub executable file specified by the "STUB" option must not be the same as the executable file being generated.

4.3.12.21 The SYMFILE Option

The "SYMFILE" option provides a method for specifying an alternate file for debugging information. The format of the "SYMFILE" option (short form "SYMF") is as follows.

OPTION SYMFILE[=symbol_file]

where:	*description:*
symbol_file	is a file specification for the name of the symbol file. If no file extension is specified, a file extension of "sym" is assumed.

By default, no symbol file is generated; debugging information is appended at the end of the executable file. Specifying this option causes the WATCOM Linker to generate a symbol file. The symbol file contains the debugging information generated by the WATCOM Linker when the "DEBUG" directive is used. The symbol file can then be used by VIDEO. If no debugging information is requested, no symbol file is created, regardless of the presence of the "SYMFILE" option.

If no file name is specified, the symbol file will have a default file extension of "sym" and the same file name as the executable file. Note that the symbol file will be created in the current

directory even if the executable file name specified in the "NAME" directive contains a path specification.

Alternatively, a file name can be specified. The following directive instructs the WATCOM Linker to generate a symbol file and call it "myprog.sym" regardless of the name of the executable file.

```
option symf=myprog
```

You can also specify a path and/or file extension when using the "SYMFILE=" form of the "SYMFILE" option.

Note:

1. This option should be used to debug a DOS "COM" executable file. A DOS "COM" executable file must not contain any additional information other than the executable information itself since DOS uses the size of the file to determine what to load.

2. This option should be used when creating a Microsoft Windows executable file. Typically, before an executable file can be executed as a Microsoft Windows application, a resource compiler takes the Windows executable file and a resource file as input and combines them. If the executable file contains debugging information, the resource compiler will strip the debugging information from the executable file. Therefore, debugging information must not be part of the executable file created by the WATCOM Linker.

4.3.12.22 The UNDEFSOK Option

The "UNDEFSOK" option tells the WATCOM Linker to generate an executable file even if undefined symbols are present. By default, no executable file will be generated if undefined symbols are present.

The format of the "UNDEFSOK" option (short form "U") is as follows.

OPTION UNDEFSOK

4.3.12.23 The VERBOSE Option

The "VERBOSE" option controls the amount of information produced by the WATCOM Linker in the map file. The format of the "VERBOSE" option (short form "V") is as follows.

OPTION VERBOSE

If the "VERBOSE" option is specified, the WATCOM Linker will list, for each object file, all segments it defines and their sizes. By default, this information is not produced in the map file.

Note that the "VERBOSE" option cannot be used in conjunction with the "MODTRACE" or "SYMTRACE" directives.

4.3.12.24 The VERSION Option

This option is allowed only when generating an OS/2 32-bit linear executable.

The "VERSION" option can be used to identify the application so that it can be distinguished from other versions (releases) of the same application. This option is most useful when creating a DLL since applications that use the DLL may only execute with a specific version of the DLL.

The format of the "VERSION" option (short form "VER") is as follows.

OPTION VERSION=major[.minor]

where:	***description:***
major	specifies the major version number.
minor	specifies the minor version number and must be less than 100.

4.3.13 The PATH Directive

The "PATH" directive is used to specify the directories that are to be searched for object files appearing in subsequent "FILE" directives. When the "PATH" directive is specified, the current directory will no longer be searched unless it appears in the "PATH" directive. Also, a "PATH" directive with no list of directories will cause the current directory to be searched again for object files. The format of the "PATH" directive (short from "P") is as follows.

PATH [path_name{;path_name}]

where: ***description:***

path_name is a path name.

Consider a directive file containing the following linker directives.

```
path \math
file sin
path \stats
file mean, variance
```

It instructs the WATCOM Linker to process the following object files:

```
\math\sin.obj
\stats\mean.obj
\stats\variance.obj
```

It is also possible to specify a list of paths in a "PATH" directive. Consider the following example.

```
path \math;\stats
file sin
```

First, the WATCOM Linker will attempt to load the file "\math\sin.obj". If unsuccessful, the WATCOM Linker will attempt to load the file "\stats\sin.obj".

It is possible to override the path specified in a "PATH" directive by preceding the object file name in a "FILE" directive with an absolute path specification. An absolute path specification is one that begins with a drive specification or the "\" character.

```
path \math
file sin
path \stats
file mean, \mydir\variance
```

The above directive file instructs the WATCOM Linker to process the following object files:

```
\math\sin.obj
\stats\mean.obj
\mydir\variance.obj
```

4.3.14 The SEGMENT Directive

The "SEGMENT" directive is used to describe the attributes of code and data segments. The format of the "SEGMENT" directive (short form "SE") is as follows.

SEGMENT seg_desc{,seg_desc}

seg_desc ::= seg_id {seg_attrs}+

seg_id ::= 'seg_name' | CLASS 'class_name'

seg_attrs ::= PRELOAD | LOADONCALL
| IOPL | NOIOPL
| EXECUTEONLY | EXECUTEREAD
| READONLY | READWRITE
| SHARE | NONSHARED
| CONFORMING | NONCONFORMING

where: ***description:***

seg_name is the name of the code or data segment whose attributes are being specified.

class_name is a class name. The attributes will be assigned to all segments belonging to the specified class.

PRELOAD (short form "PR") specifies that the segment is loaded as soon as the executable file is loaded. This is the default.

LOADONCALL (short form "LO") specifies that the segment is loaded only when accessed.

IOPL (short form "I") specifies that the segment requires I/O privilege. That is, they can access the hardware directly.

NOIOPL (short form "NOI") specifies that the segment does not require I/O privilege. This is the default.

EXECUTEONLY (short form "EXECUTEO") specifies that the segment can only be executed. This attribute should only be specified for code segments. This attribute should not be specified if it is possible for the code segment to contain jump tables which is the case with the WATCOM C and FORTRAN optimizing compilers.

EXECUTEREAD (short form "EXECUTER") specifies that the segment can only be executed and read. This attribute, the default for code segments, should only be specified for code segments. This attribute is appropriate for code segments that contain jump tables as is possible with the WATCOM C and FORTRAN optimizing compilers.

READONLY (short form "READO") specifies that the segment can only be read. This attribute should only be specified for data segments.

READWRITE (short form "READW") specifies that the segment can be read and written. This is the default for data segments. This attribute should only be specified for data segments.

SHARE (short form "SH") specifies that a single copy of the segment will be loaded and will be shared by all processes.

NONSHARED (short form "NONS") specifies that a unique copy of the segment will be loaded for each process. This is the default.

CONFORMING (short form "CON") specifies that the segment will assume the I/O privilege of the segment that referenced it. By default, the segment is "NONCONFORMING".

NONCONFORMING (short form "NONC") specifies that the segment will not assume the I/O privilege of the segment that referenced it. This is the default.

> ***Note:*** Attributes specified for segments identified by a segment name override attributes specified for segments identified by a class name.

4.3.15 The SYMTRACE Directive

The "SYMTRACE" directive instructs the WATCOM Linker to print a list of all modules that reference the specified symbols. The format of the "SYMTRACE" directive (short form "SYMT") is as follows.

> ***SYMTRACE symbol_name{,symbol_name}***

where: ***description:***

symbol_name is the name of a symbol.

The information is displayed in the map file. Consider the following example.

Example:

```
wlink form os2 op map f test l math symt sin, cos
```

The WATCOM Linker will list, in the map file, all modules that reference the symbols "sin" and "cos".

Note:

1. The "SYMTRACE" directive cannot be used in conjunction with the "VERBOSE" option.

4.3.16 The SYSTEM Directive

There are two forms of the "SYSTEM" directive.

The first form of the "SYSTEM" directive allows you to associate a set of linker directives with a user-specified name called the *system name*. The format of this form of the "SYSTEM" directive (short form "SYS") is as follows.

SYSTEM BEGIN system_name {directives} END

where: ***description:***

system_name is a user-defined system name.

directives are linker directives.

A restricted set of linker directives is allowed in a system definition. They are as follows.

- LIBFILE
- LIBPATH
- LIBRARY
- NAME
- OPTION
- SEGMENT
- SYSTEM

When using a "SYSTEM" directive in a system definition, you can only use the second form; you cannot define a system within a system definition.

The second form of the "SYTEM" directive is as follows.

SYSTEM system_name

where: ***description:***

system_name is a user-defined system name.

When this form of the "SYSTEM" directive is encountered, all directives specified in the system definition identified by `system_name` will be processed.

Let us consider an example that demonstrates the use of the "SYSTEM" directive. The following linker directives define a system called *statistics*.

```
system begin statistics
form os2
libpath \libs
library stats, graphics
option stack=8k
end
```

They specify that a *statistics* application is to be created by using the libraries "stats.lib" and "graphics.lib". These library files are located in the directory "\libs". The application requires a stack size of 8k and an PenPoint executable will be generated.

Suppose the linker directives in the above example are contained in the file "stats.lnk". If we wish to create a *statistics* application, we can issue the following command.

```
wlink @stats file myappl
```

As demonstrated by the above example, the "SYSTEM" directive can be used to localize the common attributes that describe a class of applications.

For additional examples on the use of the "SYSTEM" directive, examine the contents of the file "wlink.lnk", located in the "binb" directory of the directory in which you installed the software. If this file is located in one of the paths specified in the "PATH" environment variable, the linker will process all directives contained in the file, allowing you to reference any of the system definitions. Note that this file references the "WATCOM" environment variable which must be set to the directory in which you installed your software.

4.3.17 The # Directive

The "#" directive is used to mark the start of a comment. All text from the "#" character to the end of the line is considered a comment. The format of the "#" directive is as follows.

 # comment

where:	***description:***
comment	is any sequence of characters.

The following directive file illustrates the use of comments.

```
file main, trigtest

# Use my own version of "sin" instead of the
# library version.

file mysin
library \math\trig
```

4.3.18 The @ Directive

The "@" directive instructs the WATCOM Linker to process directives from an alternate source. The format of the "@" directive is as follows.

@directive_var   ***or***   ***@directive_file***

where: ***description:***

directive_var is the name of an environment variable. The directives specified by the value of `directive_var` will be processed.

directive_file is a file specification for the name of a linker directive file. A file extension of "lnk" is assumed if no file extension is specified.

The environment variable approach to specifying linker directives allows you to specify commonly used directives without having to specify them each time you invoke the WATCOM Linker. If the environment variable "wlink" is set as in the following example,

```
set wlink=debug all option map, verbose library math
wlink @wlink
```

each time the WATCOM Linker is invoked, full debugging information will be generated, a verbose map file will be created, and the library file "math.lib" will be searched for undefined references.

A linker directive file is useful, for example, when linker input consists of a large number of object files and you do not want to type their names on the command line each time you link your program. Note that a linker directive file can also include other linker directive files.

Let the file "memos.lnk" be a directive file containing the following lines.

```
form os2
name memos
file memos
file actions
file read
file msg
file prompt
file memmgr
library \termio\screen
library \termio\keyboard
```

Consider the following example.

Example:

```
wlink @memos
```

The WATCOM Linker is instructed to process the contents of the directive file "memos.lnk". The executable image file will be called "memos.exe". The following object files will be loaded from the current directory.

```
memos.obj
actions.obj
read.obj
msg.obj
prompt.obj
memmgr.obj
```

If any unresolved symbol references remain after all object files have been processed, the library files "screen.lib" and "keyboard.lib" in the directory "\termio" will be searched (in the order listed).

Notes:

1. In the above example, we did not provide the file extension when the directive file was specified. The WATCOM Linker assumes a file extension of "lnk" if none is present.

2. It is not necessary to list each object file and library with a separate directive. The following linker directive file is equivalent.

```
form os2
name memos
file memos,actions,read,msg,prompt,memmgr
library \termio\screen,\termio\keyboard
```

However, if you want to selectively specify what debugging information should be included, the first style of directive file will be easier to use. This is illustrated in the following sample directive file.

```
form os2
name memos
debug lines
file memos
debug all
file actions
debug lines
file read
file msg
file prompt
file memmgr
debug
library \termio\screen
library \termio\keyboard
```

3. Information for a particular directive can span directive files. This is illustrated in the following sample directive file.

```
form os2
file memos, actions, read, msg, prompt, memmgr
file @dbgfiles
library \termio\screen
library \termio\keyboard
```

The directive file "dbgfiles.lnk" contains, for example, those object files that are used for debugging purposes.

4.4 Memory Layout

The following describes the memory layout of an application linked by the WATCOM Linker. Note that this assumes that the "DOSSEG" option has been specified.

1. all segments not belonging to group "DGROUP" with class "CODE"

2. all other segments not belonging to group "DGROUP"

3. all segments belonging to group "DGROUP" with class "BEGDATA"

4. all segments belonging to group "DGROUP" not with class "BEGDATA", "BSS" or "STACK"

5. all segments belonging to group "DGROUP" with class "BSS"

6. all segments belonging to group "DGROUP" with class "STACK"

A special segment belonging to class "BEGDATA" is defined when linking with WATCOM run-time libraries. This segment is initialized with the hexadecimal byte pattern "01" and is the first segment in group "DGROUP" so that storing data at location 0 can be detected.

Segments belonging to class "BSS" contain uninitialized data. Note that this only includes uninitialized data in segments belonging to group "DGROUP". Segments belonging to class "STACK" are used to define the size of the stack used for your application. Segments belonging to the classes "BSS" and "STACK" are last in the segment ordering so that uninitialized data need not take space in the executable file.

4.5 The WATCOM Linker Memory Requirements

The WATCOM Linker uses all available memory when linking an application. For DOS-hosted versions of the WATCOM Linker, this includes expanded memory (EMS) and extended memory. It is possible for the size of the image being linked to exceed the amount of memory available in your machine, particularly if the image file is to contain debugging information. For this reason, a temporary disk file is used when all available memory is used by the WATCOM Linker.

Normally, the temporary file is created in the default directory. However, by defining the "TMP" environment variable to be a directory, you can tell the WATCOM Linker where to create the temporary file. This can be particularly useful if you have a RAM disk. Consider the following definition of the "TMP" environment variable.

```
set tmp=\tmp
```

The WATCOM Linker will create the temporary file in the directory "\tmp".

4.6 Converting Microsoft Response Files to WATCOM Linker Directive Files

A utility called MS2WLINK can be used to convert Microsoft linker response files to WATCOM Linker directive files. Input to MS2WLINK is processed in the same way as the Microsoft linker processes its input, the difference being MS2WLINK lists the corresponding WATCOM Linker directive file to the standard output device instead of a creating an

executable file. The resulting output can be redirected to a disk file which can then be used as input to the WATCOM Linker to produce an executable file.

Suppose you have a Microsoft linker response file called "test.rsp". You can convert this file to a WATCOM Linker directive file by issuing the following command.

Example:

```
ms2wlink @test.rsp >test.lnk
```

You can now use the WATCOM Linker to link your program by issuing the following command.

Example:

```
wlink @test
```

An alternative way to link your application with the WATCOM Linker from a Microsoft response file is to issue the following command.

Example:

```
ms2wlink @test.rsp | wlink
```

Since the WATCOM Linker gets its input from the standard input device, you do not have to create a WATCOM Linker directive file to link your application.

Note that MS2WLINK can also process module-definition files used for creating OS/2 and PenPoint applications.

4.7 WATCOM Linker Error Messages

The WATCOM Linker issues three classes of messages; fatal errors, errors and warnings. Each message has a 4-digit number associated with it. Fatal messages start with the digit 3, error messages start with the digit 2, and warning messages start with the digit 1. It is possible for a message to be issued as a warning or an error.

If a fatal error occurs, the linker will terminate immediately and no executable file will be generated.

If an error occurs, the linker will continue to execute so that all possible errors are issued. However, no executable file will be generated since these errors do not permit a proper executable file to be generated.

If a warning occurs, the linker will continue to execute. A warning message is usually informational and does not prevent the creation of a proper executable file. However, all warnings should eventually be corrected.

The messages listed contain references to `%s`, `%S`, `%a`, `%x`, `%d`, and `%l`. They represent strings that are substituted by the WATCOM Linker to make the error message more precise.

1. `%s` represents a string. This may be a segment or group name, or the name of a linker directive or option.

2. `%S` represents the name of a symbol.

3. `%a` represents an address. The format of the address depends on the format of the executable file being generated.

4. `%x` represents a hexadecimal number.

5. `%d` represents integers in the range -32768 and 32767.

6. `%l` represents integers in the range -2147483648 and 2147483647.

The following is a list of all warning and error messages produced by the WATCOM Linker followed by a description of the message. A message may contain more than one reference to "%s". In such a case, the description will reference them as "%sn" where n is the occurrence of "%s" in the message.

MSG 2002 **** internal ** - %s**

If this message occurs, you have found a bug in the WATCOM Linker and should contact WATCOM.

MSG 2008 **cannot open %s : %s**

An error occurred while trying to open the file "%s1". The reason for the error is given by "%s2". Generally this error message is issued when the WATCOM Linker cannot open an object file or directive file.

MSG 3009 **dynamic memory exhausted**

The WATCOM Linker uses all available memory when linking an application. For DOS-hosted versions of the WATCOM Linker, this includes expanded memory (EMS) and extended memory. When all available memory is used, a spill file will be used. Therefore, unless you are low on disk space, the WATCOM Linker will always be able to generate the executable file. Dynamic

memory is the memory the WATCOM Linker uses to build its internal data structures and symbol table. Dynamic memory is the amount of conventional memory (below 1 megabyte) available on your machine; a spill file is not used for dynamic memory. If the WATCOM Linker issues this message, it cannot link your application. The following are suggestions that may help you in this situation.

1. Concatenate all your object files into one and specify only the resulting object file as input to the WATCOM Linker. For example, if you are linking in a DOS environment, you can issue the following DOS command.

   ```
   C>copy/b *.obj all.obj
   ```

   This significantly reduces the size of the file list the WATCOM Linker must maintain.

2. Object files may contain a record which specifies the module name. This information is used by VIDEO to locate modules during a debugging session and usually contains the full path of the source file. This can consume a significant amount of memory when many such object files are being linked. If your source is being compiled by the WATCOM C compiler, you can use the "nm" option to set the module name to just the file name. This reduces the amount of memory required by the WATCOM Linker. If your are using VIDEO to debug your application, you may have to use the "set source" command so that the source corresponding to a module can be located.

3. Typically, when you are compiling a program for a large code model, each module defines a different "text" segment. If you are compiling your application using the WATCOM C compiler, you can reduce the number of "text" segments that the WATCOM Linker has to process by specifying the "nt" option. The "nt" option allows you to specify the name of the "text" segment so that a group of object files define the same "text" segment.

***MSG 2010,3010* I/O error processing %s : %s**

An error has occurred while processing the file "%s1". The cause of the error is given by "%s2". This error is usually detected while reading from object and library files or writing to the spill file or executable file. For example, this error would be issued if a "disk full" condition existed.

***MSG 2011* invalid object file attribute**

The WATCOM Linker encountered an object file that was not of the format required of an object file.

MSG 2012 **invalid library file attribute**

The WATCOM Linker encountered a library file that was not of the format required of a library file.

MSG 3013 **break key detected**

The linking process was interrupted by the user from the keyboard.

MSG 1014 **stack segment not found**

The WATCOM Linker identifies the stack segment by a segment defined as having the "STACK" attribute. This message is issued if no such segment is encountered. This usually happens if the WATCOM Linker cannot find the run-time libraries required to link your application.

MSG 2015 **bad location specified in FIXUP**

This message is issued if a bad object file is encountered.

MSG 2016 **%a: absolute target invalid for self-relative relocation**

This message is issued, for example, if a near call or jump is made to an external symbol which is defined using the "EQU" assembler directive. "%a" identifies the location of the near call or jump instruction.

MSG 2017 **bad location specified for self-relative relocation at %a**

This message is issued if a bad fixup is encountered. "%a" defines the location of the fixup.

MSG 2018 **relocation offset at %a is out of range**

This message is issued when the offset part of a relocation exceeds 64K. "%a" defines the location of the fixup. The error is most commonly caused by errors in coding assembly language routines. Consider a module that references an external symbol that is defined in a segment different from the one in which the reference occurred. The module, however, specifies that the segment in which the symbol is defined is the same segment as the segment that references the symbol. This error is most commonly caused when the "EXTRN" assembler directive is placed after the "SEGMENT" assembler directive for the segment

referencing the symbol. If the segment that references the symbol is allocated far enough away from the segment that defines the symbol, the WATCOM Linker will issue this message.

MSG 1019 **segment relocation at %a**

This message is issued when a segment relocation is encountered and "FORMAT DOS COM", "FORMAT PHARLAP" or "FORMAT NOVELL" has been specified. None of the above executable file formats allow segment relocation. "%a" identifies the location of the segment relocation.

MSG 2020 **size of group %s exceeds 64k by %l bytes**

The group "%s" has exceeded the maximum size (64k) allowed for a group by "%l" bytes. Usually, the group is "DGROUP" (the default data segment) and your application has placed too much data in this group. One of the following may solve this problem.

1. If you are using the WATCOM C compiler, you can place some of your data in a far segment by using the "far" keyword when defining data. You can also decrease the value of the data threshold by using the "zt" compiler option. Any datum whose size exceeds the value of the data threshold will be placed in a far segment.

2. If you are using the WATCOM FORTRAN 77 compiler, you can decrease the value of the data threshold by using the "dt" compiler option. Any datum whose size exceeds the value of the data threshold will be placed in a far segment.

MSG 2021 **size of segment %s exceeds 64k by %l bytes**

The segment "%s" has exceeded the maximum size (64k) of a segment. This usually occurs if you are linking an application that has been compiled for a small code model and the size of the application has grown in such a way that the size of the code segment ("_TEXT") has exceeded 64k. You can overlay your application or compile it for a large code model if you cannot reduce the amount of code in your application.

MSG 2022 **cannot have a starting address with an imported symbol**

When generating an OS/2 or PenPoint executable file, a symbol imported from a DLL cannot be a start address. When generating a NetWare 386 executable file, a symbol imported from an NLM cannot be a start address.

MSG 1023 **no starting address found, using %a**

The starting address defines the location where execution is to begin and must be defined by a special "module end" record in one of the object files linked into your application. This message is issued if no such record is encountered in which case a default starting address, namely "%a", will be used. This usually happens if the WATCOM Linker cannot find the run-time libraries required to link your application.

MSG 2024 **missing overlay loader**

This message is issued when an overlayed application is being linked and the overlay manager has not been encountered. This usually happens if the WATCOM Linker cannot find the run-time libraries required to link your application.

MSG 2025 **short vector %d is out of range**

This message is issued when the WATCOM Linker is creating an overlayed application and "OPTION SMALL" is specified. Since an overlay vector contains a near call to the overlay loader followed by a near jump to the routine corresponding to the overlay vector, all code including the overlay manager and all overlay vectors must be less than 64K. This message is issued if the offset of an overlay vector from the overlay loader or the corresponding routine exceeds 64K.

MSG 2026 **redefinition of reserved symbol %s**

The WATCOM Linker defines certain reserved symbols. These symbols are "_edata", "_end", "__OVLTAB__", "__OVLSTARTVEC__", "__OVLENDVEC__", "__LOVLLDR__", "__NOVLLDR__", "__SOVLLDR__", "__LOVLINIT__", "__NOVLINIT__" and "__SOVLINIT__". The symbols "__OVLTAB__", "__OVLSTARTVEC__", "__OVLENDVEC__", "__LOVLLDR__", "__NOVLLDR__", "__SOVLLDR__", "__LOVLINIT__", "__NOVLINIT__" and "__SOVLINIT__" are defined only if you are using overlays. The symbols "_edata" and "_end" are defined only if the "DOSSEG" option is specified. Your application must not attempt to define these symbols. "%s" identifies the reserved symbol.

MSG 1027 **redefinition of %S ignored**

The symbol "%S" has been defined by more that one module; the first definition is used. This is only a warning message. Note that if a symbol is defined more

than once and its address is the same in both cases, no warning will be issued. This prevents the warning message from being issued when linking FORTRAN 77 modules that contain common blocks.

MSG 1028 **%S is an undefined reference**

The symbol "%S" has been referenced but not defined. Check that the spelling of the symbol is consistent. If you wish the linker to ignore undefined references, use the "UNDEFSOK" option.

MSG 2029 **premature end of file encountered**

This error is issued while processing object files and object modules from libraries and is caused if the end of the file or module is reached before the "module end" record is encountered. The probable cause is a truncated object file.

MSG 2030 **multiple starting addresses found**

The starting address defines the location where execution is to begin and is defined by a "module end" record in a particular object file. This message is issued if more than one object file contains a "module end" record that defines a starting address.

MSG 2031 **segment %s is in group %s and group %s**

The segment "%s1" has been defined to be in group "%s2" in one module and in group "%s3" in another module. A segment can only belong to one group.

MSG 1032 **record (type 0x%x) not processed**

An object record type not supported by the WATCOM Linker has been encountered. This message is issued when linking object modules created by other compilers or assemblers that create object files with records that the WATCOM Linker does not support.

MSG 2033,3033 **directive error near '%s'**

A syntax error occurred while the WATCOM Linker was processing directives. "%s" specifies where the error occurred.

MSG 2034 **%a cannot have an offset with an imported symbol**

An imported symbol is one that was specified in an "IMPORT" directive. For example, under OS/2 or PenPoint, imported symbols are defined in DLLs. References to imported symbols must always have an offset value of 0. If "DosWrite" is an imported symbol, then referencing "DosWrite+2" is illegal. "%a" defines the location of the illegal reference.

MSG 1038 **DEBUG directive appears after object files**

This message is issued if the first "DEBUG" directive appears after a "FILE" directive. A common error is to specify a "DEBUG" directive after the "FILE" directives in which case no debugging information for those object files is generated in the executable file.

MSG 2039 **ALIGNMENT value too small**

The value specified in the "ALIGNMENT" option refers to the alignment of segments in the executable file. Segments in the executable file are pointed to by a segment table. An entry in the segment table contains a 16-bit value which is a multiple of the alignment value. Together they form the offset of the segment from the start of the segment table. The smaller the alignment, the bigger the value required in the segment table to point to the segment. If this value exceeds 64k, then a larger alignment value is required to decrease the size that goes in the segment table.

MSG 2040 **ordinal in IMPORT directive not valid**

The specified ordinal in the "IMPORT" directive is incorrect (e.g., -1). An ordinal number must be in the range 0 to 65535.

MSG 2041 **ordinal in EXPORT directive not valid**

The specified ordinal in the "EXPORT" directive is incorrect (e.g., -1). An ordinal number must be in the range 0 to 65535.

MSG 2042 **too many IOPL words in EXPORT directive**

The maximum number of IOPL words is 63.

MSG 1043 **duplicate exported ordinal**

This message is issued for ordinal numbers specified in an "EXPORT" directive for symbols belonging to OS/2 or PenPoint DLLs. This message is issued if an ordinal number is assigned to two different symbols. A warning is issued and

the WATCOM Linker assigns a non-used ordinal number to the symbol that caused the warning.

MSG 1044,2044 **exported symbol %s not found**

This message is issued when generating OS/2 or PenPoint DLLs and NetWare 386 NLMs. An attempt has been made to define an entry point into a DLL or NLM that does not exist.

MSG 1045 **segment attribute defined more than once**

A segment appearing in a "SEGMENT" directive has been given conflicting or duplicate attributes.

MSG 1046 **segment name %s not found**

The segment name specified in a "SEGMENT" directive has not been defined.

MSG 1047 **class name %s not found**

The class name specified in a "SEGMENT" directive has not been defined.

MSG 1048 **inconsistent attributes for automatic data segment**

This message is issued for OS/2 or PenPoint executable files. Two conflicting attributes were specified for the automatic data segment. For example, "LOADONCALL" and "PRELOAD" are conflicting attributes. Only the first attribute is used.

MSG 2049 **invalid STUB file**

The stub file is not a valid executable file. The stub file is only used for OS/2 executable files.

MSG 1050 **invalid DLL specified in OLDLIBRARY option**

The OS/2 or PenPoint DLL specified in an "OLDLIBRARY" option is not a valid dynamic link library.

MSG 2051 **STUB file name same as executable file name**

When generating an OS/2 executable file, the stub file name must not be same as the executable file name.

MSG 2052 **relocation at %a not in the same segment**

This message is only issued for protected-mode applications. A relative fixup must relocate to the same segment. "%a" defines the location of the fixup.

MSG 2053 **%a: cannot reach a DLL with a relative relocation**

A reference to a symbol in an OS/2 or PenPoint DLL must not be relative. "%a" defines the location of the reference.

MSG 1054 **debugging information incompatible: using line numbers only**

An attempt has been made to link an object file with out-of-date debugging information.

MSG 2055 **%a: frame must be the same as the target in protect mode**

Each relocation consists of three components; the location being relocated, the target (or address being referenced), and the frame (the segment to which the target is adjusted). In protected mode, the segment of the target must be the same as the frame. "%a" defines the location of the fixup.

MSG 2056 **cannot find library member %s(%s)**

Library member "%s2" in library file "%s1" could not be found. This message is issued if the library file could not be found or the library file did not contain the specified member.

MSG 3057 **executable format has been established**

This message is issued if there is more than one "FORMAT" directive.

MSG 1058 **%s option not valid for %s executable**

The option "%s1" can only be specified if an executable file whose format is "%s2" is being generated.

MSG 1059,2059 **value for %s too large**

The value specified for option "%s" exceeds its limit.

MSG 1060 **value for %s incorrect**

The value specified for option "%s" is not in the allowable range.

MSG 1061 **multiple values specified for REALBREAK**

The "REALBREAK" option can only be specified once.

MSG 1062 **DLL COMENT record invalid when not in OS2**

A DLL COMENT record contains information for symbols that are defined in dynamic link libraries. This message is issued if a reference to a DLL is encountered and the executable file format is not "OS2".

MSG 2063 **invalid relocation for flat memory model at %a**

A segment relocation in the flat memory model was encountered. "%a" defines the location of the fixup.

MSG 2064 **cannot combine 32-bit segments with 16-bit segments**

A 16-bit segment and a 32-bit segment have been encountered. Mixing object files created by an 80286 compiler and object files created by an 80386 compiler is the most probable cause of this error.

MSG 2065 **REALBREAK symbol %s not found**

The symbol specified in the "REALBREAK" option has not been defined.

MSG 2066 **invalid relative relocation type for an import at %a**

This message is issued only if a NetWare 386 executable file is being generated. An imported symbol is one that was specified in an "IMPORT" directive. Any reference to an imported symbol must not refer to the segment of the imported symbol. "%a" defines the location of the reference.

MSG 2067 **%a: cannot relocate between code and data in Novell formats**

This message is issued only if a NetWare 386 executable file is being generated. Segment relocation is not permitted. "%a" defines the location of the fixup.

MSG 2068 **absolute segment fixup not valid in protect mode**

A reference to an absolute location is not allowed in protected mode. A protected-mode application is one that is being generated for OS/2, PenPoint, Phar Lap's 386|DOS-Extender or Novell's NetWare 386 operating system. An absolute location is most commonly defined by the "EQU" assembler directive.

MSG 1069 **unload CHECK procedure not found**

The symbol specified in the "CHECK" option has not been defined.

MSG 2070 **START procedure not found**

The symbol specified in the "START" option has not been defined. The default "START" symbol is "_Prelude".

MSG 2071 **EXIT procedure not found**

The symbol specified in the "EXIT" option has not been defined. The default "STOP" symbol is "_Stop".

MSG 1072 **SECTION directive not allowed in root**

"SECTION" directives must appear between a "BEGIN" directive and its corresponding "END" directive.

MSG 2073 **bad Novell file format specified**

An invalid NetWare 386 executable file format was specified. Valid formats are NLM, DSK, NAM and LAN.

MSG 2074 **invalid NLM description**

The description specified in the "FORMAT NOVELL" directive has not been, or is incorrectly, specified.

MSG 2075 **expecting an END directive**

A "BEGIN" directive is missing its corresponding "END" directive.

MSG 1076 **%s option multiply specified**

The option "%s" can only be specified once.

MSG 2080 **file %s is a 32-bit object file**

An object file compiled for a 32-bit system has been encountered and the executable file format is a DOS "COM" file or an OS/2 1.x executable file.

MSG 2082 **invalid record type 0x%x**

An object record type not recognized by the WATCOM Linker has been encountered. This message is issued when linking object modules created by other compilers or assemblers that create object files with records that the WATCOM Linker does not recognize.

MSG 2083 **cannot reference address %a from frame %x**

The offset of a referenced symbol is greater than 64k from the location referencing it.

MSG 2084 **target offset exceeds 64K at %a**

The computed offset for a symbol exceeds 64k. "%a" defines the location of the fixup.

MSG 2086 **invalid starting address for .COM file**

The value of the segment of the starting address for a DOS "COM" file, as specified in the map file, must be 0.

MSG 1087 **stack segment ignored in .COM file**

A stack segment must not be defined when generating a DOS "COM" file. Only a single physical segment is allowed in a DOS "COM" file. The stack is allocated from the high end of the physical segment. That is, the initial value of SP is hexadecimal fffe.

MSG 3088 **virtual memory exhausted**

This message is similar to the "dynamic memory exhausted" message. The WATCOM Linker has run out of memory trying to keep track of virtual memory blocks. Virtual memory blocks are allocated from expanded memory, extended memory and the spill file.

MSG 2089 **program too large for a .COM file**

The total size of a DOS "COM" program must not exceed 64K. That is, the total amount of code and data must be less than 64K since only a single physical segment is allowed in a DOS "COM" file. You must decrease the size of your program or generate a DOS "EXE" file.

MSG 1090 **redefinition of %s by %s ignored**

The symbol "%s1" has been redefined by module "%s2". This message is issued when the size specified in the "NAMELEN" option has caused two symbols to map to the same symbol. For example, if the symbols *routine1* and *routine2* are encountered and "OPTION NAMELEN=7" is specified, then this message will be issued since the first seven characters of the two symbols are identical.

MSG 2091 **group %s is in more than one overlay**

A group that spans more than one section has been detected.

MSG 2092 **NEWSEGMENT directive appears before object files**

The "NEWSEGMENT" option must appear after a "FILE" directive.

MSG 2093 **cannot open %s**

This message is issued when the WATCOM Linker is unable to open a file and is unable to determine the cause.

MSG 2094 **i/o error processing %s**

This message is issued when the WATCOM Linker has encountered an i/o error while processing the file and is unable to determine the cause. This message may be issued when reading from object and library files, or writing to the executable and spill file.

MSG 1095 **%s debugging information too large**

The limit on the amount of the specified class of debugging information has been exceeded. The amount of line numbering information for a module cannot exceed 64k. No line numbering information for that module will be generated in the executable file if this limit is exceeded. The amount of local symbol information for a module cannot exceed 64k. No local symbol information for that module will be generated in the executable file if this limit is exceeded. The amount of typing information for a module cannot exceed 64k. No typing information for that module will be generated in the executable file if this limit is exceeded. Global symbol information is limited only by available memory.

MSG 3096 **out of symbol table space**

This message is similar to the "dynamic memory exhausted" message. The WATCOM Linker has run out of memory trying to allocate space for its symbol table.

MSG 3097 **too many library modules**

This message is similar to the "dynamic memory exhausted" message. This message if issued when the "DISTRIBUTE" option is specified. The WATCOM Linker has run out of memory trying to keep track of the relationship between object modules extracted from libraries and the overlays they should be placed in.

MSG 1098 **Phar Lap offset option must be a multiple of 4K**

The value specified with the "OFFSET" option must be a multiple of 4096.

MSG 1101 **cannot use both option verbose and a trace directive**

The "VERBOSE" option cannot be used in conjunction with the "SYMTRACE" and "MODTRACE" directives.

MSG 1102 **object file %s not found for tracing**

A "SYMTRACE" or "MODTRACE" directive contained an object file (namely %s) that could not be found.

MSG 1103 **library module %s(%s) not found for tracing**

A "SYMTRACE" or "MODTRACE" directive contained an object module (namely module %s1 in library %s2) that could not be found.

MSG 1105 **cannot reserve %l bytes of extra overlay space**

The value specified with the "AREA" option results in an executable file that requires more than 1 megabyte of memory to execute.

MSG 1107 **undefined system name: %s**

The name %s was referenced in a "SYSTEM" directive but never defined by a system block definition.

MSG 1108 **system %s defined more than once**

The name %s has appeared in a system definition block more than once.

MSG 3109 **system block %s too large**

The largest size of a system definition block is 4K. This message will be issued if the total amount of text specified in the system definition block exceeds this limit.

MSG 1110 **library members not allowed in libfile**

Only object files are allowed in a "LIBFILE" directive. This message will be issued if a module from a library file is specified in a "LIBFILE" directive.

MSG 1111 **error in default system block**

The default system block definition (system name "286" for 16-bit applications) and (system name "386" for 32-bit applications) contains a directive error. The system name "286" or "386" is automatically referenced by the linker when the format of the executable cannot be determined (i.e. no "FORMAT" directive has been specified).

MSG 3114 **environment name specified incorrectly**

This message is specified if the environment variable is not properly enclosed between two percent (%) characters.

MSG 1115 **environment name %s not found**

The environment variable %s has not been defined in the environment space.

MSG 1116 **overlay area must be at least %l bytes**

This message is issued if the size of the largest overlay exceeds the size of the overlay area specified by the "AREA" option.

MSG 1117 **segment number too high for a movable entry point**

The segment number of a moveable segment must not exceed 255. Reduce the number of segments or use the "PACKCODE" option.

MSG 1118 **heap size too large**

This message is issued if the size of the heap, stack and the default data segment (group DGROUP) exceeds 64K.

MSG 2120 **application too large to run under DOS**

This message is issued if the size of the DOS application exceeds 1M.

MSG 1121 **'%s' has already been exported**

The linker has detected an attempt to export a symbol more than once. For example, a name appearing in more than one "EXPORT" directive will cause this message to be issued. Also, if you have declared a symbol as an export in your source and have also specified the same symbol in an "EXPORT" directive, this message will be issued. This message is only a warning.

MSG 3122 **no FILE directives found**

This message is issued if no "FILE" directive has been specified. In other words, you have specified no object files to link.

MSG 3123 **OS/2 offset option must be a multiple of 64K**

The value specified with the "OFFSET" option for OS/2 2.x must be a multiple of 64K.

WATCOM Tools

5 The WATCOM Library Manager

5.1 Introduction

The WATCOM Library Manager can be used to create and update object library files. It takes as input an object file or a library file and creates or updates a library file. For OS/2 and PenPoint applications, it can also create import libraries from dynamic link libraries.

An object library is essentially a collection of object files. These object files generally contain utility routines that can be used as input to the WATCOM Linker to create an application. The following are some of the advantages of using library files.

1. Only those modules that are referenced will be included in the executable file. This eliminates the need to know which object files should be included and which ones should be left out when linking an application.

2. Libraries are a good way of organizing object files. When linking an application, you need only list one library file instead of several object files.

The WATCOM Library Manager currently runs under the following operating systems.

- DOS
- OS/2
- QNX

5.2 The WATCOM Library Manager Command Line

The following describes the WATCOM Library Manager command line.

WLIB [options_1] lib_file [options_2] [cmd_list]

The square brackets [ ] denote items which are optional. Items separated by an or-bar | and enclosed in parentheses ( ) indicate that one of the items must be specified.

lib_file is the file specification for the library file to be processed. If no file extension is specified, a file extension of "lib" is assumed.

options_1 is a list of valid options. Options may be specified in any order. If you are using a DOS or OS/2-hosted version of the WATCOM Library Manager, options are preceded by a "/" or "—" character. If you are using a QNX-hosted version of the WATCOM Library Manager, options are preceded by a "—" character.

options_2 is a list of valid options. These options are only permitted if you are running a DOS or OS/2-hosted version of the WATCOM Library Manager and must be preceded by a "/" character. The "—" character cannot be used as an option delimiter for options following the library file name since it will be interpreted as a delete command.

cmd_list is a list of commands to the WATCOM Library Manager specifying what operations are to be performed. Each command in *cmd_list* is separated by a space.

The following is a summary of valid options.

b suppress creation of backup file

c perform case sensitive comparison

d=<output_directory> directory in which extracted object modules will be placed

i(r|n)(n|o) imports for the resident/non-resident names table are to be imported by name/ordinal.

l[=<list_file>] create a listing file

n always create a new library

o=<output_file> set output file name

p=<record_size> set library record size

q operate quietly

s strip line number records from object files

t remove path information from module name specified in THEADR records

x extract all object modules from library

The following sections describe the operations that can be performed on a library file. Note that before making a change to a library file, the WATCOM Library Manager makes a backup copy of the original library file unless the "o" option is used to specify an output library file whose name is different than the original library file, or the "b" option is used to suppress the creation of the backup file. The backup copy has the same file name as the original library file but has a file extension of "bak". Hence, *lib_file* should not have a file extension of "bak".

5.3 Adding Modules to a Library File

An object file can be added to a library file by specifying a **+obj_file** command where **obj_file** is the file specification for an object file. If you are using a DOS or OS/2-hosted version of the WATCOM Library Manager, a file extension of "obj" is assumed if none is specified. If you are using a QNX-hosted version of the WATCOM Library Manager, a file extension of "o" is assumed if none is specified. If the library file does not exist, a warning message will be issued and the library file will be created.

Example:

```
wlib mylib +myobj
```

In the above example, the object file "myobj" is added to the library file "mylib.lib".

When a module is added to a library, the WATCOM Library Manager will issue a warning if a symbol redefinition occurs. This will occur if a symbol in the module being added is already defined in another module that already exists in the library file. Note that the module will be added to the library in any case.

It is also possible to combine two library files together. The following example adds all modules in the library "newlib.lib" to the library "mylib.lib".

Example:

```
wlib mylib +newlib.lib
```

Note that you must specify the "lib" file extension. Otherwise, the WATCOM Library Manager will assume you are adding an object file.

5.4 Deleting Modules from a Library File

A module can be deleted from a library file by specifying a **-mod_name** command where **mod_name** is the file name of the object file when it was added to the library with the directory and file extension removed.

Example:

```
wlib mylib -myobj
```

In the above example, the WATCOM Library Manager is instructed to delete the module "myobj" from the library file "mylib.lib".

It is also possible to specify a library file instead of a module name.

Example:

```
wlib mylib -oldlib.lib
```

In the above example, all modules in the library file "oldlib.lib" are removed from the library file "mylib.lib". Note that you must specify the "lib" file extension. Otherwise, the WATCOM Library Manager will assume you are removing an object module.

5.5 Replacing Modules in a Library File

A module can be replaced by specifying a **-+mod_name** or **+-mod_name** command. The module **mod_name** is deleted from the library. The object file "mod_name" is then added to the library.

Example:

```
wlib mylib -+myobj
```

In the above example, the module "myobj" is replaced by the object file "myobj".

It is also possible to merge two library files.

Example:

```
wlib mylib -+updlib.lib
```

In the above example, all modules in the library file "updlib.lib" replace the corresponding modules in the library file "mylib.lib". Any module in the library "updlib.lib" not in library "mylib.lib" is added to the library "mylib.lib". Note that you must specify the "lib" file extension. Otherwise, the WATCOM Library Manager will assume you are removing an object module.

5.6 Extracting a Module from a Library File

A module can be extracted from a library file by specifying a ***mod_name** command for a DOS or OS/2-hosted version of the WATCOM Library Manager or a **:mod_name** command for a QNX-hosted versions of the WATCOM Library Manager. The module **mod_name** is not deleted but is copied to a disk file. If **mod_name** is preceded by a path specification, the output file will be placed in the directory identified by the path specification.

Example:

```
wlib mylib *myobj                    DOS or OS/2-hosted
    or
wlib mylib :myobj                    QNX-hosted
```

In the above example, the module "myobj" is copied to a disk file. The disk file will be an object file with file name "myobj". If you are running a DOS or OS/2-hosted version of the WATCOM Library Manager, a file extension of "obj" will be used. If you are running a QNX-hosted version of the WATCOM Library Manager, a file extension of "o" will be used.

The specification of "mod_name" may include a directory and/or file extension. The module name is formed from the file name part of the file specification. The output file is "mod_name".

Example:

```
wlib mylib *myobj.out
```

In the above example, the module "myobj" will be extracted from the library file "mylib.lib" and placed in the file "myobj.out"

You can extract a module from a file and have that module deleted from the library file by specifying a ***-mod_name** command for a DOS or OS/2-hosted version of the WATCOM Library Manager or a **:-mod_name** command for a QNX-hosted versions of the WATCOM Library Manager. The following example performs the same operations as in the previous example but, in addition, the module is deleted from the library file.

Example:

```
wlib mylib *-myobj                    DOS or OS/2-hosted
    or
wlib mylib :-myobj                    QNX-hosted
```

Note that the same result is achieved if the delete operation precedes the extract operation.

5.7 Creating Import Libraries

The WATCOM Library Manager can also be used to create import libraries from dynamic link libraries. Import libraries are used when linking OS/2 or PenPoint applications. For more information on import and dynamic link libraries, see the chapter entitled "Linking PenPoint Applications".

Example:

```
wlib implib +dynamic.dll
```

In the above example, the following actions are performed. For each external symbol in the specified dynamic link library, a special object module is created that identifies the external symbol and the actual name of the dynamic link library it is defined in. This object module is then added to the specified library. The resulting library is called an import library.

Note that you must specify the "dll" file extension. Otherwise, the WATCOM Library Manager will assume you are adding an object file.

5.8 Creating Import Library Entries

An import library entry can be created and added to a library by specifying a command of the following form.

```
++sym.dll_name[.ordinal][.export_name]
```

where:	*description:*
sym	is the name of a symbol in a dynamic link library.
dll_name	is the name of the dynamic link library that defines `sym`.
ordinal	is the ordinal value that can be used to identify `sym` instead of using the name `export_name`. The default export name is `sym`.
export_name	is the name that an application that is linking to the dynamic link library uses to reference `sym`.

Example:

```
wlib math ++__sin.trig.1.sin
```

In the above example, an import library entry will be created for symbol `sin` and added to the library "math.lib". The symbol `sin` is defined in the dynamic link library called "trig.dll" as `__sin`. When an application is linked with the library "math.lib", the resulting executable file will contain an import by ordinal value 1. If the ordinal value was omitted, the resulting executable file would contain an import by name `sin`.

5.9 Commands from a File

The WATCOM Library Manager can be instructed to process all commands in a disk file by specifying the **@cmd_file** command where **cmd_file** is a file specification for the command file. A file extension of "lbc" is assumed if none is specified. The commands must be one of those previously described.

Example:

```
wlib mylib @mycmd
```

In the above example, all commands in the file "mycmd.lbc" are processed by the WATCOM Library Manager.

5.10 WATCOM Library Manager Options

The following sections describe the list of options allowed when invoking the WATCOM Library Manager.

5.10.1 Suppress Creation of Backup File - "b" Option

The "b" option tells the WATCOM Library Manager to not create a backup library file. In the following example, the object file identified by "new" will be added to the library file "mylib.lib".

Example:

```
wlib -b mylib +new
```

If the library file "mylib.lib" already exits, no backup library file ("mylib.bak") will be created.

5.10.2 Case Sensitive Symbol Names - "c" Option

The "c" option tells the WATCOM Library Manager to use a case sensitive compare when comparing a symbol to be added to the library to a symbol already in the library file. This will cause the names "myrtn" and "MYRTN" to be treated as different symbols. By default, comparisons are case insensitive. That is the symbol "myrtn" is the same as the symbol "MYRTN".

5.10.3 Specify Output Directory - "d" Option

The "d" option tells the WATCOM Library Manager the directory in which all extracted modules are to be placed. The default is to place all extracted modules in the current directory.

In the following example, the module "mymod" is extracted from the library "mylib.lib". If you are running a DOS or OS/2-hosted version of the WATCOM Library Manager, the

module will be placed in the file "\obj\mymod.obj". If you are running a QNX-hosted version of the WATCOM Library Manager, the module will be placed in the file "/o/mymod.o".

Example:

```
wlib -d=\obj mymod          DOS or OS/2-hosted
     or
wlib -d=/o mymod            QNX-hosted
```

5.10.4 Generating Imports from Dynamic Link Libraries - "i" Option

When creating import libraries from dynamic link libraries, import entries for the names in the resident and non-resident names tables are created. The "i" option can be used to describe the method used to import these names.

Specifying "iro" causes imports for names in the resident names table to be imported by ordinal. Specifying "irn" causes imports for names in the resident names table to be imported by name. This is the default. Specifying "ino" causes imports for names in the non-resident names table to be imported by ordinal. This is the default. Specifying "inn" causes imports for names in the non-resident names table to be imported by name.

Example:

```
wlib -iro -inn implib +dynamic.dll
```

Note that you must specify the "dll" file extension for the dynamic link library. Otherwise an object file will be assumed.

5.10.5 Creating a Listing File - "l" Option

The "l" (lower case "L") option instructs the WATCOM Library Manager to produce a list of the names of all symbols that can be found in the library file to a listing file. The file name of the listing file is the same as the file name of the library file. The file extension of the listing file is "lst".

Example:

```
wlib -l mylib
```

In the above example, the WATCOM Library Manager is instructed to list the contents of the library file "mylib.lib" and produce the output to a listing file called "mylib.lst".

An alternate form of this option is `-l=list_file`. With this form, you can specify the name of the listing file. When specifying a listing file name, a file extension of "lst" is assumed if none is specified.

Example:

```
wlib -l=mylib.out mylib
```

In the above example, the WATCOM Library Manager is instructed to list the contents of the library file "mylib.lib" and produce the output to a listing file called "mylib.out".

You can get a listing of the contents of a library file to the terminal by specifying only the library name on the command line as demonstrated by the following example.

Example:

```
wlib mylib
```

5.10.6 Always Create a New Library - "n" Option

The "n" option tells the WATCOM Library Manager to always create a new library file. That is, if the library file already exists, it becomes the backup library (unless the "b" option was specified), the original contents are discarded and a new library is created. If the "n" option was not specified, the existing library would be updated.

Example:

```
wlib -n mylib +myobj
```

In the above example, a library file called "mylib.lib" is created. It will contain a single object module, namely "myobj", regardless of the contents of "mylib.lib" prior to issuing the above command. If "mylib.lib" already exists, it will be renamed to "mylib.bak".

5.10.7 Specifying an Output File Name - "o" Option

The "o" option can be used to specify the output library file name if you want the original library to remain unchanged and a new library created.

Example:

```
wlib -o=newlib lib1 +lib2.lib
```

In the above example, the modules from "lib1.lib" and "lib2.lib" are added to the library "newlib.lib". Note that since the original library remains unchanged, no backup copy is created. Also, if the "l" option is used to specify a listing file, the listing file will assume the file name of the output library.

5.10.8 Specifying a Library Record Size - "p" Option

The "p" option specifies the record size in bytes for each record in the library file. The record size must be a power of 2 and in the range 16 to 32768. If the record size is less than 16, it will be rounded up to 16. If the record size is greater than 16 and not a power of 2, it will be rounded up to the nearest power of 2. The default record size is 16 bytes.

Each entry in the dictionary of a library file contains an offset from the start of the file which points to a module. The offset is 16 bits and is a multiple of the record size. Since the default record size if 16, the maximum size of a library file for a record size of 16 is 16*64K. If the size of the library file increases beyond this size, you must increase the record size.

Example:

```
wlib -p=32 lib1 +lib2.lib
```

In the above example, the WATCOM Library Manager is instructed to create/update the library file "lib1.lib" by adding the modules from the library file "lib2.lib". The record size of the resulting library file is 32 bytes.

5.10.9 Operate Quietly - "q" Option

The "q" option suppressing the banner and copyright notice that is normally displayed when the WATCOM Library Manager is invoked.

Example:

```
wlib -q -l mylib
```

5.10.10 Strip Line Number Records - "s" Option

The "s" option tells the WATCOM Library Manager to remove line number records from object files that are being added to a library. Line number records are generated in the object file if the "d1" option is specified when compiling the source code.

Example:

```
wlib -s mylib +myobj
```

5.10.11 Trim Module Name - "t" Option

The "t" option tells the WATCOM Library Manager to remove path information from the module name specified in THEADR records in object files that are being added to a library. The module name is created from the file name by the compiler and placed in the THEADR record of the object file. The module name will contain path information if the file name given to the compiler contains path information,

Example:

```
wlib -t mylib +myobj
```

5.10.12 Explode Library File - "x" Option

The "x" option tells the WATCOM Library Manager to extract all modules from the library. Note that the modules are not deleted from the library. Object modules will be placed in the current directory unless the "d" option is used to specify an alternate directory.

In the following example all modules will be extracted from the library "mylib.lib" and placed in the current directory.

Example:

```
wlib -x mylib
```

In the following example all modules will be extracted from the library "mylib.lib". If you are running a DOS or OS/2-hosted version of the WATCOM Library Manager, the module will be placed in the "\obj" directory. If you are running a QNX-hosted version of the WATCOM Library Manager, the module will be placed in the file "/o" directory.

Example:

```
wlib -x -d=\obj mylib          DOS or OS/2-hosted
    or
wlib -x -d=/o mylib            QNX-hosted
```

5.11 WATCOM Library Manager Error Messages

The following messages may be issued by the WATCOM Library Manager.

Error! Could not open object file '%s'.
Object file '%s' could not be found. This message is usually issued when an attempt is made to add a non-existent object file to the library.

Error! Could not open library file '%s'.
The specified library file could not be found. This is usually issued for input library files. For example, if you are combining two library files, the library file you are adding is an input library file and the library file you are adding to or creating is an output library file.

Error! Invalid object module in file '%s' not added.
The specified file contains an invalid object module.

Error! Dictionary too large. Recommend split library into two libraries.
The size of the dictionary in a library file cannot exceed 64K. You must split the library file into two separate library files.

Error! Redefinition of module '%s' in file '%s'.
This message is usually issued when an attempt is made to add a module to a library that already contains a module by that name.

Warning! Redefinition of symbol '%s' in file '%s' ignored.
This message is issued if a symbol defined by a module already in the library is also defined by a module being added to the library.

Error! Library too large. Recommend split library into two libraries or try a larger page_bound than %xH.
The record size of the library file does not allow the library file to increase beyond its current size. The record size of the library file must be increased using the "p" option.

Error! Expected '%s' in '%s' but found '%s'.
An error occurred while scanning command input.

Warning! Could not find module '%s' for deletion.
This message is issued if an attempt is made to delete a module that does not exist in the library.

Error! Could not find module '%s' for extraction.
This message is issued if an attempt is made to extract a module that does not exist in the library.

Error! Could not rename old library for backup.
The WATCOM Library Manager creates a backup copy before making any changes (unless the "b" option is specified). This message is issued if an error occurred while trying to rename the original library file to the backup file name.

Warning! Could not open library '%s' : will be created.
The specified library does not exist. It is usually issued when you are adding to a non-existent library. The WATCOM Library Manager will create the library.

Warning! Output library name specification ignored.
This message is issued if the library file specified by the "o" option could not be opened.

Warning! Could not open library '%s' and no operations specified: will not be created.
This message is issued if the library file specified on the command line does not exist and no operations were specified. For example, asking for a listing file of a non-existent library will cause this message to be issued.

Warning! Could not open listing file '%s'.
The listing file could not be opened. For example, this message will be issued when a "disk full" condition is present.

Error! Could not open output library.
The output library could not be opened.

Error! Unable to write to output library.
An error occurred while writing to the output library.

Error! Unable to write to extraction file '%s'.
This message is issued when extracting an object module from a library file and an error occurs while writing to the output file.

Error! Out of Memory.
There was not enough memory to process the library file.

Error! Could not open file '%s'.
This message is issued if the output file for a module that is being extracted from a library could not be opened.

Error! Library '%s' is invalid. Contents ignored.
The library file does not contain the correct header information.

Error! Library '%s' has an invalid page size. Contents ignored.
The library file has an invalid record size. The record size is contained in the library header and must be a power of 2.

Error! Invalid object record found in file '%s'.
The specified file contains an invalid object record.

Error! No library specified on command line.
This message is issued if a library file name is not specified on the command line.

Error! Expecting library name.
This message is issued if the location of the library file name on the command line is incorrect.

Warning! Invalid file name '%s'.
This message is issued if an invalid file name is specified. For example, a file name longer that 127 characters is not allowed.

Error! Could not open command file '%s'.
The specified command file could not be opened.

Error! Could not read from file '%s'. Contents ignored as command input.
An error occurred while reading a command file.

6 The WATCOM Strip Utility

6.1 Introduction

Once a program has been debugged, the WATCOM Strip Utility allows you to remove the debugging information from the executable file so that you do not have to remove the debugging directives from the linker directive file and link your program again. Removal of the debugging information reduces the size of the executable image.

All executable files generated by the WATCOM Linker can be specified as input to the WATCOM Strip Utility. Note that for executable files created for Novell's NetWare 386 operating system, debugging information created using the "NOVELL" option in the "DEBUG" directive cannot be removed from the executable file. You must remove the "DEBUG" directive from the directive file and re-link your application.

The WATCOM Strip Utility currently runs under the following operating systems.

- DOS
- OS/2
- QNX

6.2 The WATCOM Strip Utility Command Line

The WATCOM Strip Utility command line syntax is:

WSTRIP exe_file [output_file] [symbol_file]

The square brackets [ ] denote items which are optional.

exe_file is a file specification for the name of an executable file. If no file extension is specified, the WATCOM Strip Utility will assume one of the following extensions: "exe", "dll", "exp", "rex", "nlm", "dsk", "lan", "nam", "qnx" or no file extension. Note that the order specified in the list of file extensions is the order in which the WATCOM Strip Utility will select file extensions.

output_file is an optional file specification for the output file. If no file extension is specified, the file extension specified in the input file name will be used for the output file name.

symbol_file is an optional file specification for the file in which the symbolic debugging information is to be stored. If no file extension is specified, a file extension of "sym" is assumed. To specify the name of the symbol file but not the name of the output file, a "." may be specified in place of *output_file*.

Description:

1. If *output_file* is not specified, the debugging information is removed from *exe_file*.

2. If *output_file* is specified, *exe_file* is copied to *output_file* and the debugging information is removed from *output_file*. *exe_file* remains unchanged.

3. If *symbol_file* is specified then the debugging information that is removed from the executable file is placed in this file. The symbolic debugging information may be put back on to the executable by concatenating the debugging information to the end of the executable file.

4. During processing, the WATCOM Strip Utility will create a temporary file, ensuring that a file by the chosen name does not already exist.

6.3 WATCOM Strip Utility Error Messages

The following error messages may be issued by the WATCOM Strip Utility.

Usage: WSTRIP input_file
or: WSTRIP input_file output_file [symbol_file]
or: WSTRIP input_file . [symbol_file]

The command line was entered incorrectly.

Too low on memory
There is not enough free memory to allocate file buffers.

Unable to find '%s'
The specified file could not be located.

Cannot create temporary file
All the temporary file names are in use.

Unable to open '%s' to read
The input executable file cannot be opened for reading.

'%s' is not a valid executable file
The input file has invalid executable file header information.

'%s' does not contain debugging information
There is nothing to strip from the specified executable file.

Seek error on '%s'
An error occurred during a seek operation on the specified file.

Unable to create output file '%s'
The output file could not be created. Check that the output disk is not write-protected or that the specified output file is not marked "read-only".

Unable to create symbol file '%s'
The symbol file could not be created.

Error reading '%s'
An error occurred while reading the input executable file.

Error writing to '%s'
An error occurred while writing the output executable file or the symbol file. Check the amount of free space on the output disk. If the input and output files reside on the same disk, there might not be enough room for a second copy of the executable file during processing.

Cannot erase file '%s'
The input executable file is probably marked "read-only" and therefore could not be erased (the input file is erased whenever the output file has the same name).

Cannot rename file '%s'

The output executable file could not be renamed. Ordinarily, this should never occur.

7 The Make Utility

7.1 Introduction

The WATCOM Make utility is useful in the development of programs and text processing but is general enough to be used in many different applications. Make uses the fact that each file has a time-stamp associated with it that indicates the last time the file was updated. Make uses this time-stamp to decide which files are out of date with respect to each other. For instance, if we have an input data file and an output report file we would like the output report file to accurately reflect the contents of the input data file. In terms of time-stamps, we would like the output report to have a more recent time-stamp than the input data file (we will say that the output report file should be "younger" than the input data file). If the input file had been modified then we would know from the younger time-stamp (in comparison to the report file) that the report file was out of date and should be updated. Make may be used in this and many other situations to ensure that files are kept up to date.

Some readers will be quite familiar with the concepts of the Make file maintenance tool. WATCOM Make is patterned after the Make utility found on UNIX systems. The next major section is simply intended to summarize, for reference purposes only, the syntax and options of Make's command line and special macros. Subsequent sections go into the philosophy and capabilities of WATCOM Make. If you are not familiar with the capabilities of the Make utility, we recommend that you skip to the next major section entitled "Dependency Declarations" and read on.

The sample code in `\PENPOINT\SDK\SAMPLE` contains WATCOM Make files for all sample applications. You can refer to these Make files for examples of how to write WATCOM Make files.

7.2 WATCOM Make Reference

7.2.1 WATCOM Make Command Line Format

The formal WATCOM Make command line syntax is shown below.

WMAKE [options] [macro_defs] [targets]

As indicated by the square brackets [ ], all items are optional.

options is a list of valid WATCOM Make options, each preceded by a slash ("/") or a dash ("–"). Options may be specified in any order.

macro_defs is a list of valid WATCOM Make macro definitions. Macro definitions are of the form:

```
A=B
```

and are readily identified by the presence of the "=" (the "#" character may be used instead of the "=" character if necessary). Surround the definition with quotes (") if it contains blanks (e.g., "debug_opt=debug all"). The macro definitions specified on the command line supersede any macro definitions defined in makefiles.

targets is one or more targets described in the makefile.

7.2.2 WATCOM Make Options Summary

In this section, we present a terse summary of the WATCOM Make options. This summary is displayed on the screen by simply entering "WMAKE ?" on the command line.

Example:

```
C>wmake ?
```

/a make all targets by ignoring time-stamps
/b block/ignore all implicit rules
/c do not verify the existence of files made
/d debug mode - echo all work as it progresses
/e always erase target after error/interrupt (disables prompting)
/f the next parameter is a name of dependency description file
/h do not print out Make identification lines (no header)
/i ignore return status of all commands executed

/k	on error/interrupt: continue on next target
/l	the next parameter is the name of a output log file
/m	do not search for MAKEINIT file
/n	no execute mode - print commands without executing
/o	use circular implicit rule path
/p	print the dependency tree as understood from the file
/q	query mode - check targets without updating them
/r	do not use default definitions
/s	silent mode - do not print commands before execution
/t	touch files instead of executing commands
/u	UNIX compatibility mode
/z	do not erase target after error/interrupt (disables prompting)

7.2.3 Command Line Options

Command line options, available with Make, allow you to control the processing of the makefile.

Options

/a make all targets by ignoring time-stamps

The "a" option is a safe way to update every target. For program maintenance, it is the preferred method over deleting object files or touching source files.

/b block/ignore all implicit rules

The "b" option will indicate to Make that you do not want any implicit rule checking done. The "b" option is useful in makefiles containing double colon "::" explicit rules because an implicit rule search is conducted after a double colon "::" target is updated. Including the directive `.BLOCK` in a makefile also will disable implicit rule checking.

/c do not verify the existence of files made

Make will check to ensure that a target exists after the associated command list is executed. The target existence checking may be disabled with the "c" option. The "c" option is useful in processing makefiles that were developed with other Make utilities. The `.NOCHECK` directive may be used to disable target existence in a makefile.

/d debug mode - echo all work as it progresses

The "d" option will print out information about the time-stamp of files and indicate how the makefile processing is proceeding.

/e always erase target after error/interrupt (disables prompting)

The "e" option will indicate to Make that, if an error or interrupt occurs during makefile processing, the current target being made may be deleted without prompting. The `.ERASE` directive may be used as an equivalent option in a makefile.

/f the next parameter is a name of dependency description file

The "f" option specifies that the next parameter on the command line is the name of a makefile which must be processed. If the "f" option is specified then the search for the default makefile named "MAKEFILE" is not done. Any number of makefiles may be processed with the "f" option.

Example:

```
wmake /f myfile
wmake /f myfile1 /f myfile2
```

/h do not print out Make identification lines (no header)

The "h" option is useful for less verbose output. Combined with the "q" option, this allows a batch file to silently query if an application is up to date. Combined with the "n" option, a batch file could be produced containing the commands necessary to update the application.

/i ignore return status of all commands executed

The "i" option is equivalent to the `.IGNORE` directive.

/k on error/interrupt: continue on next target

Make will stop updating targets when a non-zero status is returned by a command. The "k" option will continue processing targets that do not depend on the target that caused the error. The `.CONTINUE` directive in a makefile will enable this error handling capability.

/l the next parameter is the name of a output log file

Make will output an error message when a non-zero status is returned by a command. The "l" option specifies a file that will record all error messages output by Make during the processing of the makefile.

/m do not search for the MAKEINIT file

The default action for Make is to search for an initialization file called "MAKEINIT". The "m" option will indicate to Make that processing of the MAKEINIT file is not desired.

/n no execute mode - print commands without executing

The "n" option will print out what commands should be executed to update the application without actually executing them. Combined with the "h" option, a batch file could be produced which would contain the commands necessary to update the application.

Example:

```
wmake /h /n >update.bat
update
```

This is useful for applications which require all available resources (memory and devices) for executing the updating commands.

/o use circular implicit rule path

When this option is specified, Make will use a circular path specification search which may save on disk activity for large makefiles. The "o" option is equivalent to the `.OPTIMIZE` directive.

/p print out makefile information

The "p" option will cause Make to print out information about all the explicit rules, implicit rules, and macro definitions.

/q query mode - check targets without updating them

The "q" option will cause Make to return a status of 1 if the application requires updating; it will return a status of 0 otherwise. Here is a example batch file using the "q" option:

Example:

```
wmake /q
if errorstatus 0 goto noupdate
wmake /q /h /n >\tmp\update.bat
call \tmp\update.bat
:noupdate
```

/r do not use default definitions

The default definitions are:

```
__MAKEOPTS__ = <options passed to WMAKE>
__MAKEFILES__ = <list of makefiles>
__MSDOS__ =
# clear .EXTENSIONS list
.EXTENSIONS:

# set .EXTENSIONS list
.EXTENSIONS: .exe .exp .lib .obj .asm        &
             .c .for .pas .cob .h .fi .mif
```

For OS/2, the `__MSDOS__` macro will be replaced by `__OS2__`. The "r" option will disable these definitions before processing any makefiles.

/s silent mode - do not print commands before execution

The "s" option is equivalent to the `.SILENT` directive.

/t touch files instead of executing commands

Sometimes there are changes which are purely cosmetic (adding a comment to a source file) that will cause targets to be updated needlessly thus wasting computer resources. The "t" option will make files appear younger without altering their contents. The "t" option is useful but should be used with caution.

/u UNIX compatibility mode

The "u" option will indicate to Make that the line continuation character should be a backslash "\" rather than an ampersand "&".

/z do not erase target after error/interrupt (disables prompting)

The "z" option will indicate to Make that if an error or interrupt occurs during makefile processing then the current target being made should not be deleted. The `.HOLD` directive in a makefile has the same effect as the "z" option.

7.2.4 Special Macros

Make has many different special macros. Here are some of the simpler ones.

Macro	*Expansion*
$$	represents the character "$"
$#	represents the character "#"
$@	full file name of the target
$*	target with the extension removed
$<	list of all dependents
$?	list of dependents that are younger than the target

The following macros are for more sophisticated makefiles.

Macro	*Expansion*
__MSDOS__	This macro is defined in the MS/DOS environment.
__OS2__	This macro is defined in the OS/2 environment.
__MAKEOPTS__	contains all of the command line options that WMAKE was invoked with except for the "f" options
__MAKEFILES__	contains the names of all of the makefiles processed at the time of expansion (includes the file currently being processed)

The next three tables contain macros that are valid during execution of command lists for explicit rules, implicit rules, and the `.ERROR` directive. The expansion is presented for the following example:

Example:

```
a:\dir\target.ext : b:\dir1\dep1.ex1 c:\dir2\dep2.ex2
```

Macro	*Expansion*
$^@	a:\dir\target.ext
$^*	a:\dir\target
$^&	target
$^.	target.ext
$^:	a:\dir\

Macro	*Expansion*
$[@	b:\dir1\dep1.ex1
$[*	b:\dir1\dep1
$[&	dep1
$[.	dep1.ex1
$[:	b:\dir1\

Macro	*Expansion*
$]@	c:\dir2\dep2.ex2
$]*	c:\dir2\dep2
$]&	dep2
$].	dep2.ex2
$]:	c:\dir2\

7.3 Dependency Declarations

In order for Make to be effective, a list of file dependencies must be declared. The declarations may be entered into a text file of any name but Make will read a file called "MAKEFILE" by default if it is invoked as follows:

Example:

```
C>wmake
```

If you want to use a file that is not called "MAKEFILE" then the command line option "f" will cause Make to read the specified file instead of the default "MAKEFILE".

Example:

```
C>wmake /f myfile
```

We will now go through an example to illustrate how Make may be used for a simple application. Suppose we have an input file, a report file, and a report generator program then we may declare a dependency as follows:

```
#
# (a comment in a makefile starts with a "#")
# simple dependency declaration
#
balance.lst : ledger.dat
        doreport
```

Note that the dependency declaration starts at the beginning of a line while commands always have at least one blank or tab before them. This form of a dependency declaration is called an *explicit rule*. The file "BALANCE.LST" is called the *target* of the rule. The *dependent* of the rule is the file "LEDGER.DAT" while "DOREPORT" forms one line of the *rule command list*. The dependent is separated from the target by a colon.

> ***Hint:*** A good habit to develop is to always put spaces around the colon so that it will not be confused with drive specifications (e.g., a:).

The explicit rule declaration indicates to Make that the program "DOREPORT" should be executed if "LEDGER.DAT" is younger than "BALANCE.LST" or if "BALANCE.LST" does not yet exist. In general, if the dependent file has a more recent modification date and time than the target file then WATCOM Make will execute the specified command.

> *Note:* The terminology employed here is used by S.I.Feldman of Bell Laboratories in *Make - A Program for Maintaining Computer Programs*. Confusion often arises from the use of the word "dependent". In this context, it means "a subordinate part". In the example, "LEDGER.DAT" is a subordinate part of the report "BALANCE.LST".

7.4 Multiple Dependents

Suppose that our report "BALANCE.LST" becomes out-of-date if any of the files "LEDGER.DAT", "SALES.DAT" or "PURCHASE.DAT" are modified. We may modify the dependency rule as follows:

```
#
# multiple dependents rule
#
balance.lst : ledger.dat sales.dat purchase.dat
        doreport
```

This is an example of a rule with multiple dependents. In this situation, the program "DOREPORT" should be executed if any of "LEDGER.DAT", "SALES.DAT" or "PURCHASE.DAT" are younger than "BALANCE.LST" or if "BALANCE.LST" does not yet exist. In cases where there are multiple dependents, if any of the dependent files has a more recent modification date and time than the target file then WATCOM Make will execute the specified command.

7.5 Multiple Targets

Suppose that the "DOREPORT" program produces two reports. If both of these reports require updating as a result of modification to the dependent files, we could change the rule as follows:

```
#
# multiple targets and multiple dependents rule
#
balance.lst summary.lst : ledger.dat sales.dat purchase.dat
        doreport
```

Suppose that you entered the command:

```
wmake
```

which causes Make to start processing the rules described in "MAKEFILE". In the case where multiple targets are listed in the makefile, Make will, by default, process only the first target it encounters. In the example, Make will check the date and time of "BALANCE.LST" against its dependents since this is the first target listed.

To indicate that some other target should be processed, the target is specified as an argument to the Make command.

Example:

```
wmake summary.lst
```

There are a number of interesting points to consider:

1. By default, Make will only check that the target file exists after the command ("DOREPORT" in this example) is executed. It does not check that the target's time-stamp shows it to be younger. If the target file does not exist after the command has been executed, an error is reported.

2. There is no guarantee that the command you have specified does update the target file. In other words, simply because you have stated a dependency does not mean that one exists.

3. Furthermore, it is not implied that other targets in our list will not be updated. In the case of our example, you can assume that we have designed the "doreport" command to update both targets.

7.6 Multiple Rules

A makefile may consist of any number of rules. Note that the following:

```
target1 target2 : dependent1 dependent2 dependent3
        command list
```

is equivalent to:

```
target1 : dependent1 dependent2 dependent3
        command list

target2 : dependent1 dependent2 dependent3
        command list
```

Also, the rules may depend on the targets of other rules.

```
#
# rule 1: this rule uses rule 2
#
balance.lst summary.lst : ledger.dat sales.dat purchase.dat
        doreport

#
# rule 2: used by rules 1 and 3
#
sales.dat : canada.dat england.dat usa.dat
        dosales

#
# rule 3: this rule uses rule 2
#
year.lst : ledger.dat sales.dat purchase.dat
        doyearly
```

The dependents are checked to see if they are the targets of any other rules in the makefile in which case they are updated. This process of updating dependents that are targets in other rules continues until a rule is reached that has only simple dependents that are not targets of rules. At this point, if the target does not exist or if any of the dependents is younger than the target then the command list associated with the rule is executed.

> ***Hint:*** The term "updating", in this context, refers to the process of checking the time-stamps of dependents and running the specified command list whenever they are out-of-date. Whenever a dependent is the target of some other rule, the dependent must be brought up-to-date first. Stated another way, if "A" depends on "B" and "B" depends on "C" and "C" is younger than "B" then we must update "B" before we update "A".

Make will check to ensure that the target exists after its associated command list is executed. The target existence checking may be disabled in two ways:

1. use the command line option "c"
2. use the `.NOCHECK` directive.

The rule checking returns to the previous rule that had the target as a dependent. Upon returning to the rule, the command list is executed if the target does not exist or if any of the updated dependents are now younger than the target. If you were to type:

```
wmake
```

here are the steps that would occur with the previous makefile:

```
update(balance.lst) (rule 1)

  update(ledger.dat)              (not a target)
  update(sales.dat)               (found rule 2)

    update(canada.dat)            (not a target)
    update(england.dat)           (not a target)
    update(usa.dat)               (not a target)
    IF sales.dat does not exist                          OR
       any of (canada.dat,england.dat,usa.dat)
         is younger than sales.dat
    THEN execute "dosales"

  update(purchase.dat)            (not a target)
  IF balance.lst does not exist                          OR
     any of (ledger.dat,sales.dat,purchase.dat)
       is younger than (balance.lst)
  THEN execute "doreport"
```

The third rule in the makefile will not be included in this update sequence of steps. Recall that the default target that is "updated" is the first target in the first rule encountered in the makefile. This is the default action taken by Make when no target is specified on the command line. If you were to type:

```
wmake year.lst
```

then the file "YEAR.LST" would be updated. As Make reads the rules in "MAKEFILE", it discovers that updating "YEAR.LST" involves updating "SALES.DAT". The update sequence is similar to the previous example.

7.7 Targets Without Any Dependents - ".SYMBOLIC"

There must always be at least one target in a rule but it is not necessary to have any dependents. If a target does not have any dependents, the command list associated with the rule will always be executed if the target is updated.

You might ask, "What may a rule with no dependents be used for?". A rule with no dependents may be used to describe actions that are useful for the group of files being maintained. Possible uses include backing up files, cleaning up files, or printing files.

To illustrate the use of the `.SYMBOLIC` directive, we will add two new rules to the previous example. First, we will omit the `.SYMBOLIC` directive and observe what will happen when it is not present.

```
#
# rule 4: backup the data files
#
backup :
        echo "insert backup disk"
        pause
        copy *.dat a:
        echo "backup complete"
#
# rule 5: cleanup temporary files
#
cleanup :
        del *.tmp
        del \tmp\*.*
```

and then execute the command:

```
wmake backup
```

Make will execute the command list associated with the "backup" target and issue an error message indicating that the file "BACKUP" does not exist after the command list was executed. The same thing would happen if we typed:

```
wmake cleanup
```

In this makefile we are using "backup" and "cleanup" to represent actions we want performed. The names are not real files but rather they are symbolic names. This special type of target may be declared with the `.SYMBOLIC` directive. This time, we show rules 4 and 5 with the appropriate addition of `.SYMBOLIC` directives.

```
#
# rule 4: backup the data files
#
backup : .SYMBOLIC
        echo "insert backup disk"
        pause
        copy *.dat a:
        echo "backup complete"
#
# rule 5: cleanup temporary files
#
cleanup : .SYMBOLIC
        del *.tmp
        del \tmp\*.*
```

The use of the `.SYMBOLIC` directive indicates to Make that the target should always be updated internally after the command list associated with the rule has been executed. A short form for the common idiom of singular `.SYMBOLIC` targets like:

```
target : .SYMBOLIC
        commands
```

is:

```
target
        commands
```

This kind of target definition is useful for many types of management tasks that can be described in a makefile.

7.8 Preserving Targets - ".PRECIOUS"

Most DOS or OS/2 utilities and programs have special return codes that indicate error conditions. Make will check the return code for every command executed. If the return code is non-zero, Make will stop processing the current rule and optionally delete the current target being updated. If a file is precious enough that this treatment of return codes is not wanted then the `.PRECIOUS` directive may be used. The `.PRECIOUS` directive indicates to Make that the target should not be deleted if an error occurs during the execution of the associated command list. Here is an example of the `.PRECIOUS` directive:

```
#
# .PRECIOUS example
#
balance.lst summary.lst : ledger.dat sales.dat purchase.dat .PRECIOUS
        doreport
```

If the program "DOREPORT" executes and its return code is non-zero then Make will not attempt to delete "BALANCE.LST" or "SUMMARY.LST". If only one of the files is precious then the makefile could be coded as follows:

```
#
# .PRECIOUS example
#
balance.lst : .PRECIOUS
balance.lst summary.lst : ledger.dat sales.dat purchase.dat
        doreport
```

The file "BALANCE.LST" will not be deleted if an error occurs while the program "DOREPORT" is executing.

7.9 Ignoring Return Codes - ".IGNORE"

Some programs do not have meaningful return codes so for these programs we want to ignore the return code completely. There are different ways to ignore return codes namely,

1. use the command line option "i"
2. put a "–" in front of specific commands, or
3. use the `.IGNORE` directive.

In the following example, the rule:

```
#
# ignore return code example
#
balance.lst summary.lst : ledger.dat sales.dat purchase.dat
        -doreport
```

will ignore the return status from the program "DOREPORT". Using the dash in front of the command is the preferred method for ignoring return codes because it allows Make to check all the other return codes.

The `.IGNORE` directive is used as follows:

```
#
# .IGNORE example
#
.IGNORE
balance.lst summary.lst : ledger.dat sales.dat purchase.dat
        doreport
```

Using the `.IGNORE` directive will cause Make to ignore the return code for every command. The "i" command line option and the `.IGNORE` directive prohibit Make from performing any error checking on the commands executed and, as such, should be used with caution.

Another way to handle non-zero return codes is to continue processing targets which do not depend on the target that had a non-zero return code during execution of its associated command list. There are two ways of indicating to Make that processing should continue after a non-zero return code:

1. use the command line option "k"
2. use the `.CONTINUE` directive.

7.10 Erasing Targets After Error - ".ERASE"

Most DOS or OS/2 utilities and programs have special return codes that indicate error conditions. Make will check the return code for every command executed. If the return code is non-zero, Make will stop processing the current rule and optionally delete the current target being updated. By default, Make will prompt for deletion of the current target. The `.ERASE` directive indicates to Make that the target should be deleted if an error occurs during the execution of the associated command list. No prompt is issued in this case. Here is an example of the `.ERASE` directive:

```
#
# .ERASE example
#
.ERASE
balance.lst summary.lst : ledger.dat sales.dat purchase.dat
        doreport
```

If the program "DOREPORT" executes and its return code is non-zero then Make will attempt to delete "BALANCE.LST" or "SUMMARY.LST" depending on which it was updating.

7.11 Preserving Targets After Error - ".HOLD"

Most DOS or OS/2 utilities and programs have special return codes that indicate error conditions. Make will check the return code for every command executed. If the return code is non-zero, Make will stop processing the current rule and optionally delete the current target being updated. By default, Make will prompt for deletion of the current target. The `.HOLD` directive indicates to Make that the target should not be deleted if an error occurs during the execution of the associated command list. No prompt is issued in this case. The `.HOLD` directive is similar to `.PRECIOUS` but applies to all targets listed in the makefile. Here is an example of the `.HOLD` directive:

```
#
# .HOLD example
#
.HOLD
balance.lst summary.lst : ledger.dat sales.dat purchase.dat
        doreport
```

If the program "DOREPORT" executes and its return code is non-zero then Make will not delete "BALANCE.LST" or "SUMMARY.LST".

7.12 Suppressing Terminal Output - ".SILENT"

As commands are executed, Make will print out the current command before it is executed. It is possible to execute the makefile without having the commands printed. There are three ways to inhibit the printing of the commands before they are executed, namely:

1. use the command line option "s"
2. put an "@" in front of specific commands, or
3. use the `.SILENT` directive.

In the following example, the rule:

```
#
# silent command example
#
balance.lst summary.lst : ledger.dat sales.dat purchase.dat
        @doreport
```

will prevent the string "doreport" from being printed on the screen before the command is executed.

The `.SILENT` directive is used as follows:

```
#
# .SILENT example
#
.SILENT
balance.lst summary.lst : ledger.dat sales.dat purchase.dat
        doreport
```

Using the `.SILENT` directive or the "s" command line option will inhibit the printing of all commands before they are executed.

At this point, most of the capability of Make may be realized. Methods for making makefiles more succinct will be discussed.

7.13 Macros

Make has a simple macro facility that may be used to improve makefiles by making them easier to read and maintain. A macro identifier may be composed from a string of alphabetic characters and numeric characters. The underscore character is also allowed in a macro identifier. If the macro identifier starts with a "%" character, the macro identifier represents an environment variable. For instance, the macro identifier "%path" represents the environment variable "path".

Macro identifiers Valid?

2morrow yes

stitch_in_9 yes

invalid~id no

2b_or_not_2b yes

%path yes

reports yes

!@#*% no

We will use a programming example to show how macros are used. The programming example involves four C source files and two header files. Here is the initial makefile (before macros):

```
#
# programming example
# (before macros)
#
plot.exp : main.obj input.obj calc.obj output.obj
        wlink @plot

main.obj : main.c defs.h globals.h
        wcc386 main /mf /d1 /w3

calc.obj : calc.c defs.h globals.h
        wcc386 calc /mf /d1 /w3
input.obj : input.c defs.h globals.h
        wcc386 input /mf /d1 /w3

output.obj : output.c defs.h globals.h
        wcc386 output /mf /d1 /w3
```

Macros become useful when changes must be made to makefiles. If the programmer wanted to change the compiler options for the different compiles, the programmer would have to make a global change to the makefile. With this simple example, it is quite easy to make the change but try to imagine a more complex example with different programs having similar options. The global change made by the editor could cause problems by changing the options for other programs. A good habit to develop is to define macros for any programs that have command line options. In our example, we would change the makefile to be:

```
#
# programming example
# (after macros)
#
wlink_options =
wcc_options = /mf /d1 /w3

plot.exp : main.obj input.obj calc.obj output.obj
        wlink $(wlink_options) @plot

main.obj : main.c defs.h globals.h
        wcc386 main $(wcc_options)
calc.obj : calc.c defs.h globals.h
        wcc386 calc $(wcc_options)

input.obj : input.c defs.h globals.h
        wcc386 input $(wcc_options)

output.obj : output.c defs.h globals.h
        wcc386 output $(wcc_options)
```

A macro definition consists of a macro identifier starting on the beginning of the line followed by an "=" which in turn is followed by the text to be replaced. A macro may be redefined, with the latest declaration being used for subsequent expansions (no warning is given upon redefinition of a macro). The replacement text may contain macro references.

A macro reference may occur in two forms. The previous example illustrates one way to reference macros whereby the macro identifier is delimited by "$(" and ")". The parentheses are optional so the macro "wcc_options" could be referenced by:

```
main.obj : main.c defs.h globals.h
        wcc386 main $wcc_options
```

Certain ambiguities may arise with this form of macro reference. For instance, examine this makefile fragment:

Example:

```
temporary_dir = \tmp\
temporary_file = $temporary_dirtmp000.tmp
```

The intention of the declarations is to have a macro that will expand into a file specification for a temporary file. Make will collect the largest identifier possible before macro expansion occurs. The macro reference is followed by text that looks like part of the macro identifier ("tmp000") so the macro identifier that will be referenced will be "temporary_dirtmp000". The incorrect macro identifier will not be defined so an error message will be issued.

If the makefile fragment was:

```
temporary_dir = \tmp\
temporary_file = $(temporary_dir)tmp000.tmp
```

there would be no ambiguity. The preferred way to reference macros is to enclose the macro identifier by "$(" and ")".

Macro references are expanded immediately on dependency lines (and thus may not contain references to macros that have not been defined) but other macro references have their expansion deferred until they are used in a command. In the previous example, the macros "wlink_options" and "wcc_options" will not be expanded until the commands that reference them are executed.

Another use for macros is to replace large amounts of text with a much smaller macro reference. In our example, we only have two header files but suppose we had very many header files. Each explicit rule would be very large and difficult to read and maintain. We will use the previous example makefile to illustrate this use of macros.

```
#
# programming example
# (with more macros)
#
wlink_options =
wcc_options = /mf /d1 /w3
header_files = defs.h globals.h
object_files = main.obj input.obj calc.obj &
               output.obj

plot.exp : $(object_files)
        wlink $(wlink_options) @plot

main.obj : main.c $(header_files)
        wcc386 main $(wcc_options)
calc.obj : calc.c $(header_files)
        wcc386 calc $(wcc_options)

input.obj : input.c $(header_files)
        wcc386 input $(wcc_options)

output.obj : output.c $(header_files)
        wcc386 output $(wcc_options)
```

Notice the ampersand ("&") at the end of the macro definition for "object_files". The ampersand indicates that the macro definition continues on the next line. In general, if you want to continue a line in a makefile, use an ampersand ("&") at the end of the line.

There are special macros provided by Make to access DOS or OS/2 environment variable names. To access the **PATH** environment variable in a makefile, we use the macro identifier "%path". For example, if we have the following line in a command list:

Example:

```
echo $(%path)
```

it will print out the current value of the **PATH** environment variable when it is executed.

There are two other special environment macros that are predefined by Make. The macro identifier "%cdrive" will expand into one letter representing the current drive. The macro identifier "%cwd" will expand into the current working directory. These macro identifiers are not very useful unless we can specify that they be expanded immediately. The complementary macros "$+" and "$–" respectively turn on and turn off immediate expansion of macros. The scope of the "$+" macro is the current line after which the default macro expansion behaviour is resumed. A possible use of these macros is illustrated by the following example makefile.

```
#
# $(%cdrive), $(%cwd), $+, and $- example
#
dir1 = $(%cdrive):$(%cwd)
dir2 = $+ $(dir1) $-
example : .SYMBOLIC
        cd ..
        echo $(dir1)
        echo $(dir2)
```

Which would produce the following output if the current working directory is C:\WATCOM\SOURCE\EXAMPLE:

Example:

```
(command output only)
C:\WATCOM\SOURCE
C:\WATCOM\SOURCE\EXAMPLE
```

The macro definition for "dir2" forces immediate expansion of the "%cdrive" and "%cwd" macros thus defining "dir2" to be the current directory that Make was invoked in. The macro "dir1" is not expanded until execution time when the current directory has changed from the initial directory.

Combining the $+ and $– special macros with the special macro identifiers "%cdrive" and "%cwd" is a useful makefile technique. The $+ and $– special macros are general enough to be used in many different ways.

Constructing other macros is another use for the $+ and $– special macros. Make allows macros to be redefined and combining this with the $+ and $– special macros, similar looking macros may be constructed.

```
#
# macro construction with $+ and $-
#
template = file1.$(ext) file2.$(ext) file3.$(ext) file4.$(ext)
ext = dat
data_files = $+ $(template) $-
ext = lst
listing_files = $+ $(template) $-

example : .SYMBOLIC
        echo $(data_files)
        echo $(listing_files)
```

This makefile would produce the following output:

Example:

```
file1.dat file2.dat file3.dat file4.dat
file1.lst file2.lst file3.lst file4.lst
```

Adding more text to a macro can also be done with the $+ and $– special macros.

```
#
# macro addition with $+ and $-
#
objs = file1.obj file2.obj file3.obj
objs = $+$(objs)$- file4.obj
objs = $+$(objs)$- file5.obj

example : .SYMBOLIC
        echo $(objs)
```

This makefile would produce the following output:

Example:

```
file1.obj file2.obj file3.obj file4.obj file5.obj
```

Make provides a shorthand notation for this type of macro operation. Text can be added to a macro by using the "+=" macro assignment. The previous makefile can be written as:

```
#
# macro addition with +=
#
objs  = file1.obj file2.obj file3.obj
objs += file4.obj
objs += file5.obj

example : .SYMBOLIC
        echo $(objs)
```

and still produce the same results. The shorthand notation "+=" supported by Make provides a quick way to add more text to macros.

There are instances when it is useful to have macro identifiers that have macro references contained in them. If you wanted to print out an informative message before linking the executable that was different between the debugging and production version, we would express it as follows:

```
#
# programming example
# (macro selection)
#
version = debugging           # debugging version

msg_production = linking production version ...
msg_debugging = linking debug version ...

wlink_options_production =
wlink_options_debugging = debug all
wlink_options = $(wlink_options_$(version))
wcc_options_production = /mf /w3
wcc_options_debugging = /mf /d1 /w3
wcc_options = $(wcc_options_$(version))

header_files = defs.h globals.h
object_files = main.obj input.obj calc.obj &
               output.obj

plot.exp : $(object_files)
        echo $(msg_$(version))
        wlink $(wlink_options) @plot

main.obj : main.c $(header_files)
        wcc386 main $(wcc_options)
```

```
calc.obj : calc.c $(header_files)
        wcc386 calc $(wcc_options)

input.obj : input.c $(header_files)
        wcc386 input $(wcc_options)

output.obj : output.c $(header_files)
        wcc386 output $(wcc_options)
```

Take notice of the macro references that are of the form:

```
$(<partial_macro_identifier>$(version))
```

The expansion of a macro reference begins by expanding any macros seen until a matching right parenthesis is found. The macro identifier that is present after the matching parenthesis is found will be expanded. The other form of macro reference namely:

```
$<macro_identifier>
```

may be used in a similar fashion. The previous example would be of the form:

```
$<partial_macro_identifier>$version
```

Macro expansion occurs until a character that cannot be in a macro identifier is found (on the same line as the "$") after which the resultant macro identifier is expanded. If you want two macros to be concatenated then the line would have to be coded:

```
$(macro1)$(macro2)
```

The use of parentheses is the preferred method for macro references because it completely specifies the order of expansion.

In the previous example, we can see that the four command lines that invoke the compiler are very similar in form. We may make use of these similarities by denoting the command by a macro reference. We need to be able to define a macro that will expand into the correct command when processed. Fortunately, Make can reference the first member of the dependent list, the last member of the dependent list, and the current target being updated with the use of some special macros. These special macros have the form:

```
$<file_specifier><form_qualifier>
```

where <file_specifier> is one of:

"^"	represents the current target being updated
"["	represents the first member of the dependent list
"]"	represents the last member of the dependent list

and <form_qualifier> is one of:

"@"	full file name
"*"	file name with extension removed
"&"	file name with path and extension removed
"."	file name with path removed
":"	path of file name

If the file "D:\DIR1\DIR2\NAME.EXT" is the current target being updated then the following example will show how the form qualifiers are used.

Macro	*Expansion for D:\DIR1\DIR2\NAME.EXT*
$^@	`D:\DIR1\DIR2\NAME.EXT`
$^*	`D:\DIR1\DIR2\NAME`
$^&	`NAME`
$^.	`NAME.EXT`
$^:	`D:\DIR1\DIR2\`

These special macros provide the capability to reference targets and dependents in a variety of ways.

```
#
# programming example
# (more macros)
#
version = debugging            # debugging version

msg_production = linking production version ...
msg_debugging = linking debug version ...

wlink_options_production =
wlink_options_debugging = debug all
wlink_options = $(wlink_options_$(version))
wcc_options_production = /mf /w3
wcc_options_debugging = /mf /d1 /w3
wcc_options = $(wcc_options_$(version))

wcc386_command = wcc386 $[* $(wcc_options)

header_files = defs.h globals.h
object_files = main.obj input.obj calc.obj &
               output.obj
plot.exp : $(object_files)
        echo $(msg_$(version))
        wlink $(wlink_options) @$^*

main.obj : main.c $(header_files)
        $(wcc386_command)

calc.obj : calc.c $(header_files)
        $(wcc386_command)

input.obj : input.c $(header_files)
        $(wcc386_command)

output.obj : output.c $(header_files)
        $(wcc386_command)
```

This example illustrates the use of the special dependency macros. Notice the use of "$^*" in the linker command. The macro expands into the string "plot" since "plot.exp" is the target when the command is processed. The use of the special dependency macros is recommended because they make use of information that is already contained in the dependency rule.

At this point, we know that macro references begin with a "$" and that comments begin with a "#". What happens if we want to use these characters without their special meaning? Make has two special macros that provide these characters to you. The special macro "$$" will result in a "$" when expanded and "$#" will expand into a "#". These special macros are

provided so that you are not forced to work around the special meanings of the "$" and "#" characters.

7.14 Implicit Rules

Make is capable of accepting declarations of commonly used dependencies. These declarations are called "implicit rules" as opposed to "explicit rules" which were discussed previously. Implicit rules may be applied only in instances where you are able to describe a dependency in terms of file extensions.

> ***Hint:*** Recall that a DOS or OS/2 file extension may be at most three characters and is separated from the end of the file name by a period ("."). In the DOS or OS/2 file specification:
>
> c:\dos\ansi.sys
>
> the file extension is "SYS".

An implicit rule provides a command list for a dependency between files with certain extensions. The form of an implicit rule is as follows:

```
.<dependent_extension>.<target_extension>:
        <command_list>
```

Implicit rules are used if a file has not been declared as a target in any explicit rule or the file has been declared as a target in an explicit rule with no command list. For a given target file, a search is conducted to see if there are any implicit rules defined for the target file's extension in which case Make will then check if the file with the dependent extension in the implicit rule exists. If the file with the dependent extension exists then the command list associated with the implicit rule is executed and processing of the makefile continues.

Other implicit rules for the target extension are searched in a similar fashion. The order in which the dependent extensions are checked becomes important if there is more than one implicit rule declaration for a target extension. If we have the following makefile fragment:

Example:

```
.pas.obj:
        (command list)
.c.obj:
        (command list)
```

an ambiguity arises. If we have a target file "TEST.OBJ" then which do we check for first, "TEST.PAS" or "TEST.C"? Make handles this with the `.EXTENSIONS` directive. The `.EXTENSIONS` directive declares which extensions are allowed to be used in implicit rules and how these extensions are ordered. The default `.EXTENSIONS` declaration is:

```
.EXTENSIONS:
.EXTENSIONS: .exe .exp .lib .obj .asm .c .for .pas .cob .h .fi .mif
```

A `.EXTENSIONS` directive with an empty list will clear the `.EXTENSIONS` list and any previously defined implicit rules. Any subsequent `.EXTENSIONS` directives will add extensions to the end of the list.

> ***Hint:*** The default `.EXTENSIONS` declaration could have been coded as:
>
> .EXTENSIONS:
>
> .EXTENSIONS: .exe
>
> .EXTENSIONS: .exp
>
> .EXTENSIONS: .lib
>
> .EXTENSIONS: .obj
>
> .EXTENSIONS: .asm .c .for .pas .cob .h .fi .mif
>
> with identical results.

Make will not allow any implicit rule declarations that use extensions that are not in the current `.EXTENSIONS` list. Returning to our makefile fragment:

```
.pas.obj:
        (command list)
.c.obj:
        (command list)
```

and our target file "TEST.OBJ", we now know that the `.EXTENSIONS` list determines in what order the dependents "TEST.PAS" and "TEST.C" will be tried. If the .EXTENSIONS declaration is:

Example:

```
.EXTENSIONS:
.EXTENSIONS: .exe .obj .asm .pas .c .for .cob
```

we can see that the dependent file "TEST.PAS" will be tried first as a possible dependent with "TEST.C" being tried next.

One apparent problem with implicit rules and their associated command lists is that they are used for many different targets and dependents during the processing of a makefile. The same problem occurs with commands constructed from macros. Recall that there is a set of special macros that start with "$^", "$[", or "$]" that reference the target, first dependent, or last dependent of an explicit dependency rule. In an implicit rule there may be only one dependent or many dependents depending on whether the rule is being executed for a target with a single colon ":" or double colon "::" dependency. If the target has a single colon or double colon dependency, the "$^", "$[", and "$]" special macros will reflect the values in the rule that caused the implicit rule to be invoked. Otherwise, if the target does not have a dependency rule then the "$[" and "$]" special macros will be set to the same value, namely, the file found in the implicit rule search.

We will use the last programming example to illustrate a possible use of implicit rules.

```
#
# programming example
# (implicit rules)
#
version = debugging            # debugging version

msg_production = linking production version ...
msg_debugging = linking debug version ...

wlink_options_production =
wlink_options_debugging = debug all
wlink_options = $(wlink_options_$(version))
```

```
wcc_options_production = /mf /w3
wcc_options_debugging = /mf /d1 /w3
wcc_options = $(wcc_options_$(version))

header_files = defs.h globals.h
object_files = main.obj input.obj calc.obj &
               output.obj

plot.exp : $(object_files)
        echo $(msg_$(version))
        wlink $(wlink_options) @$^*

.c.obj:
        wcc386 $[* $(wcc_options)
main.obj : main.c $(header_files)

calc.obj : calc.c $(header_files)

input.obj : input.c $(header_files)

output.obj : output.c $(header_files)
```

As this makefile is processed, any time an object file is found to be older than its associated source file or header files then Make will attempt to execute the command list associated with the explicit rule. Since there are no command lists associated with the four object file targets, an implicit rule search is conducted. Suppose "CALC.OBJ" was older than "CALC.C". The lack of a command list in the explicit rule with "CALC.OBJ" as a target causes the ".c.obj" implicit rule to be invoked for "CALC.OBJ". The file "CALC.C" is found to exist so the commands

```
wcc386 CALC /mf /d1 /w3
echo linking debug version ...
wlink debug all @plot
```

are executed. The last two commands are a result of the compilation of "CALC.C" producing a "CALC.OBJ" file that is younger than the "PLOT.EXP" file that in turn must be generated again.

The use of implicit rules is straightforward when all the files that the makefile deals with are in the current directory. Larger applications may have files that are in many different directories. Suppose we moved the programming example files to three sub-directories.

Files	*Sub-directory*
***.H**	\EXAMPLE\H
***.C**	\EXAMPLE\C
rest	\EXAMPLE\O

Now the previous makefile (located in the \EXAMPLE\O sub-directory) would look like this:

```
#
# programming example
# (implicit rules)
#
h_dir   = \example\h\ #sub-directory containing header files
c_dir   = \example\c\ #sub-directory containing C files
version = debugging   # debugging version
msg_production = linking production version ...
msg_debugging = linking debug version ...

wlink_options_production =
wlink_options_debugging = debug all
wlink_options = $(wlink_options_$(version))

wcc_options_production = /mf /w3
wcc_options_debugging = /mf /dl /w3
wcc_options = $(wcc_options_$(version))

header_files = $(h_dir)defs.h $(h_dir)globals.h
object_files = main.obj input.obj calc.obj &
               output.obj
plot.exp : $(object_files)
        echo $(msg_$(version))
        wlink $(wlink_options) @$^*

.c.obj:
        wcc386 $[* $(wcc_options)

main.obj : $(c_dir)main.c $(header_files)

calc.obj : $(c_dir)calc.c $(header_files)

input.obj : $(c_dir)input.c $(header_files)

output.obj : $(c_dir)output.c $(header_files)
```

Suppose "\EXAMPLE\O\CALC.OBJ" was older than "\EXAMPLE\C\CALC.C". The lack of a command list in the explicit rule with "CALC.OBJ" as a target causes the ".c.obj" implicit rule to be invoked for "CALC.OBJ". At this time, the file "\EXAMPLE\O\CALC.C" is not found so an error is reported indicating that "CALC.OBJ" could not be updated. How may implicit rules be useful in larger applications if they will only search the current directory for the dependent file? We must specify more information about the dependent extension (in this case ".C"). We do this by associating a path with the dependent extension as follows:

```
.<dependent_extension> : <path_specification>
```

This allows the implicit rule search to find the files with the dependent extension.

> ***Hint:*** A valid path specification is made up of directory specifications separated by semicolons (";"). Here are some path specifications:
>
> d:;c:\dos;c:\utils;c:\wc
>
> c:\sys
>
> a:\bin;d:
>
> Notice that these path specifications are identical to the form required by the DOS or OS/2 "PATH" command.

Our makefile will be correct now if we add the new declaration as follows:

```
#
# programming example
# (implicit rules)
#
h_dir   = \example\h\ #sub-directory containing header files
c_dir   = \example\c\ #sub-directory containing C files
version = debugging    # debugging version
msg_production = linking production version ...
msg_debugging = linking debug version ...

wlink_options_production =
wlink_options_debugging = debug all
wlink_options = $(wlink_options_$(version))
wcc_options_production = /mf /w3
wcc_options_debugging = /mf /dl /w3
wcc_options = $(wcc_options_$(version))

header_files = $(h_dir)defs.h $(h_dir)globals.h
object_files = main.obj input.obj calc.obj &
               output.obj

plot.exp : $(object_files)
        echo $(msg_$(version))
        wlink $(wlink_options) @$^*
```

```
.c:     $(c_dir)
.c.obj:
        wcc386 $[* $(wcc_options)

main.obj : $(c_dir)main.c $(header_files)

calc.obj : $(c_dir)calc.c $(header_files)

input.obj : $(c_dir)input.c $(header_files)

output.obj : $(c_dir)output.c $(header_files)
```

Suppose "\EXAMPLE\O\CALC.OBJ" is older than "\EXAMPLE\C\CALC.C". The lack of a command list in the explicit rule with "CALC.OBJ" as a target will cause the ".c.obj" implicit rule to be invoked for "CALC.OBJ". The dependent extension ".C" has a path associated with it so the file "\EXAMPLE\C\CALC.C" is found to exist. The commands

```
wcc386 \EXAMPLE\C\CALC /mf /d1 /w3
echo linking debug version ...
wlink debug all @plot
```

are executed to update the necessary files.

If the application requires many source files in different directories Make will search for the files using their associated path specifications. For instance, if the current example files were setup as follows:

Sub-directory Contents

\EXAMPLE\H

```
DEFS.H, GLOBALS.H
```

\EXAMPLE\C\PROGRAM

```
MAIN.C, CALC.C
```

\EXAMPLE\C\SCREEN

```
INPUT.C, OUTPUT.C
```

\EXAMPLE\O

```
PLOT.EXP, MAKEFILE, MAIN.OBJ, CALC.OBJ, INPUT.OBJ,
OUTPUT.OBJ
```

the makefile would be changed to:

```
#
# programming example
# (implicit rules)
#
h_dir         = ..\h\     # sub-directory with header files
                          # sub-directories with .C files
program_dir   = ..\c\program\ # - MAIN.C, CALC.C
screen_dir    = ..\c\screen\  # - INPUT.C, OUTPUT.C
version       = debugging     # debugging version

msg_production = linking production version ...
msg_debugging = linking debug version ...

wlink_options_production =
wlink_options_debugging = debug all
wlink_options = $(wlink_options_$(version))
wcc_options_production = /mf /w3
wcc_options_debugging = /mf /dl /w3
wcc_options = $(wcc_options_$(version))

header_files = $(h_dir)defs.h $(h_dir)globals.h
object_files = main.obj input.obj calc.obj &
               output.obj

plot.exp : $(object_files)
        echo $(msg_$(version))
        wlink $(wlink_options) @$^*

.c:     $(program_dir);$(screen_dir)
.c.obj:
        wcc386 $[* $(wcc_options)
main.obj : $(program_dir)main.c $(header_files)

calc.obj : $(program_dir)calc.c $(header_files)

input.obj : $(screen_dir)input.c $(header_files)

output.obj : $(screen_dir)output.c $(header_files)
```

Suppose that there is a change in the "DEFS.H" file which causes all the source files to be recompiled. The implicit rule ".c.obj" is invoked for every object file so the corresponding ".C" file must be found for each ".OBJ" file. We will show where Make searches for the C source files.

```
update    main.obj
  test    ..\c\program\main.c            (it does exist)
  execute wcc386 ..\c\program\main /mf /dl /w3

update    calc.obj
  test    ..\c\program\calc.c            (it does exist)
  execute wcc386 ..\c\program\calc /mf /dl /w3
```

```
update    input.obj
  test    ..\c\program\input.c        (it does not exist)
  test    ..\c\screen\input.c             (it does exist)
  execute wcc386 ..\c\screen\input /mf /dl /w3

update    output.obj
  test    ..\c\program\output.c       (it does not exist)
  test    ..\c\screen\output.c            (it does exist)
  execute wcc386 ..\c\screen\output /mf /dl /w3

etc.
```

Notice that Make checked the sub-directory "..\C\PRORGAM" for the files "INPUT.C" and "OUTPUT.C". Make optionally may use a circular path specification search which may save on disk activity for large makefiles. The circular path searching may be used in two different ways:

1. use the command line option "o"
2. use the `.OPTIMIZE` directive.

Make will retain (for each suffix) what sub-directory yielded the last successful search for a file. The search for a file is resumed at this directory in the hope that wasted disk activity will be minimized. If the file cannot be found in the sub-directory then Make will search the next sub-directory in the path specification (cycling to the first sub-directory in the path specification after an unsuccessful search in the last sub-directory).

Changing the previous example to include this feature, results in the following:

```
#
# programming example
# (optimized path searching)
#
.OPTIMIZE

h_dir         = ..\h\     # sub-directory with header files
                          # sub-directories with .C files
program_dir   = ..\c\program\ # - MAIN.C, CALC.C
screen_dir    = ..\c\screen\  # - INPUT.C, OUTPUT.C
version       = debugging     # debugging version

msg_production = linking production version ...
msg_debugging = linking debug version ...

wlink_options_production =
wlink_options_debugging = debug all
wlink_options = $(wlink_options_$(version))
```

```
wcc_options_production = /mf /w3
wcc_options_debugging = /mf /dl /w3
wcc_options = $(wcc_options_$(version))

header_files = $(h_dir)defs.h $(h_dir)globals.h
object_files = main.obj input.obj calc.obj &
               output.obj

plot.exp : $(object_files)
        echo $(msg_$(version))
        wlink $(wlink_options) @$^*
.c:     $(program_dir);$(screen_dir)
.c.obj:
        wcc386 $[* $(wcc_options)

main.obj : $(program_dir)main.c $(header_files)

calc.obj : $(program_dir)calc.c $(header_files)

input.obj : $(screen_dir)input.c $(header_files)

output.obj : $(screen_dir)output.c $(header_files)
```

Suppose again that there is a change in the "DEFS.H" file which causes all the source files to be recompiled. We will show where Make searches for the C source files using the optimized path specification searching.

```
update    main.obj
  test    ..\c\program\main.c            (it does exist)
  execute wcc386 ..\c\program\main /mf /dl /w3

update    calc.obj
  test    ..\c\program\calc.c            (it does exist)
  execute wcc386 ..\c\program\calc /mf /dl /w3
update    input.obj
  test    ..\c\program\input.c       (it does not exist)
  test    ..\c\screen\input.c            (it does exist)
  execute wcc386 ..\c\screen\input /mf /dl /w3

update    output.obj
  test    ..\c\screen\output.c           (it does exist)
  execute wcc386 ..\c\screen\output /mf /dl /w3

etc.
```

Make did not check the sub-directory "..\C\PROGRAM" for the file "OUTPUT.C" because the last successful attempt to find a ".C" file occurred in the "..\C\SCREEN" sub-directory. In this small example, the amount of disk activity saved by Make is not substantial but the savings become much more pronounced in larger makefiles.

> ***Hint:*** The simple heuristic method that Make uses for optimizing path specification searches namely, keeping track of the last successful sub-directory, is very effective in reducing the amount of disk activity during the processing of a makefile. A pitfall to avoid is having two files with the same name in the path. The version of the file that is used to update the target depends on the previous searches. Care should be taken when using files that have the same name with path specifications.

Large makefiles for projects written in C may become difficult to maintain with all the header file dependencies. Ignoring header file dependencies and using implicit rules may reduce the size of the makefile while keeping most of the functionality intact. The previous example may be made smaller by using this idea.

```
#
# programming example
# (no header dependencies)
#
.OPTIMIZE

h_dir         = ..\h\     # sub-directory with header files
                          # sub-directories with .C files
program_dir   = ..\c\program\ # - MAIN.C, CALC.C
screen_dir    = ..\c\screen\  # - INPUT.C, OUTPUT.C
version       = debugging     # debugging version

msg_production = linking production version ...
msg_debugging = linking debug version ...

wlink_options_production =
wlink_options_debugging = debug all
wlink_options = $(wlink_options_$(version))
wcc_options_production = /mf /w3
wcc_options_debugging = /mf /dl /w3
wcc_options = $(wcc_options_$(version))

object_files = main.obj input.obj calc.obj &
               output.obj

plot.exp : $(object_files)
        echo $(msg_$(version))
        wlink $(wlink_options) @$^*

.c:     $(program_dir);$(screen_dir)
.c.obj:
        wcc386 $[* $(wcc_options)
```

Implicit rules are very useful in this regard providing you are aware that you have to make up for the information that is missing from the makefile. In the case of C programs, you must ensure that you force Make to compile any programs affected by changes in header files. Forcing Make to compile programs may be done by touching source files (not recommended), deleting object files, or using the "a" option and targets on the command line. Here is how the

files "INPUT.OBJ" and "MAIN.OBJ" may be recompiled if a change in some header file affects both files.

Example:

```
del input.obj
del main.obj
wmake
```

or using the "a" option

Example:

```
wmake /a input.obj main.obj
```

The possibility of introducing bugs into programs is present when using this makefile technique because it does not protect the programmer completely from object modules becoming out-of-date. The use of implicit rules without header file dependencies is a viable makefile technique but it is not without its pitfalls.

7.15 Double Colon Explicit Rules

Single colon ":" explicit rules are useful in many makefile applications. However, the single colon rule has certain restrictions that make it difficult to express more complex dependency relationships. The restrictions imposed on single colon ":" explicit rules are:

1. only one command list is allowed for each target
2. after the command list is executed, the target is considered up to date

The first restriction becomes evident when you want to update a target in different ways (i.e., when the target is out of date with respect to different dependents). The double colon explicit rule removes this restriction.

```
#
# multiple command lists
#
target1 :: dependent1 dependent2
        command1

target1 :: dependent3 dependent4
        command2
```

Notice that if "target1" is out of date with respect to either "dependent1" or "dependent2" then "command1" will be executed. The double colon "::" explicit rule does not consider the target (in this case "target1") up to date after the command list is executed. Make will continue to

attempt to update "target1". Afterwards "command2" will be executed if "target1" is out of date with respect to either "dependent3" or "dependent4". It is possible that both "command1" and "command2" will be executed. As a result of the target not being considered up to date, an implicit rule search will be conducted on "target1" also. Make will process the double colon "::" explicit rules in the order that they are encountered in the makefile. A useful application of the double colon "::" explicit rule involves maintaining and using prototype information generated by a compiler.

```
#
# double colon "::" example
#
options = /w3

# generate macros for the .OBJ and .DEF files
template = module1.$(ext) module2.$(ext) module3.$(ext)
ext = obj
objs = $+ $(template) $-
ext = def
defs = $+ $(template) $-

# add .DEF to the extensions list
.EXTENSIONS:
.EXTENSIONS: .exe .obj .def .c
# implicit rules for the .OBJ and .DEF files
.c.obj:
        wcc386 $[* $(options)

# generate the prototype file (only do a syntax check)
.c.def:
        wcc386 $[* $(options) /v/zs

program.exp :: $(defs)
        erase *.err

program.exp :: $(objs)
        wlink @$^*
```

The ".OBJ" files are updated to complete the update of the file "PROGRAM.EXP". It is important to keep in mind that Make does not consider the file "PROGRAM.EXP" up to date until it has conducted a final implicit rule search. The double colon "::" explicit rule is useful when describing complex update actions.

7.16 Preprocessing Directives

One of the primary objectives in using a make utility is to improve the development and maintenance of projects. A programming project consisting of many makefiles in different sub-directories may become unwieldy to maintain. The maintenance problem stems from the amount of duplicated information scattered throughout the project makefiles. Make provides a method to reduce the amount of duplicated information present in makefiles. Preprocessing directives provide the capability for different makefiles to make use of common information.

7.16.1 File Inclusion

A common solution to the "duplicated information" problem involves referencing text contained in one file from many different files. Make supports file inclusion with the `!include` preprocessing directive. The development of object libraries, using WATCOM C/386, for the different 80x86 memory models provides an ideal example to illustrate the use of the `!include` preprocessing directive.

Sub-directory Contents

\WINDOW `WINDOW.CMD, WINDOW.MIF`

\WINDOW\H
`PROTO.H, GLOBALS.H, BIOS_DEF.H`

\WINDOW\C
`WINDOW.C, KEYBOARD.C, MOUSE.C, BIOS.C`

\WINDOW\SCSD
small model object files, `MAKEFILE, WINDOW_S.LIB`

\WINDOW\SCBD
compact model object files, `MAKEFILE, WINDOW_C.LIB`

\WINDOW\BCSD
medium model object files, `MAKEFILE, WINDOW_M.LIB`

\WINDOW\BCBD
large model object files, `MAKEFILE, WINDOW_L.LIB`

\WINDOW\BCHD
huge model object files, `MAKEFILE, WINDOW_L.LIB`

The WLIB command file "WINDOW.CMD" contains the list of library operations required to build the libraries. The contents of "WINDOW.CMD" are:

```
-+window
-+bios
-+keyboard
-+mouse
```

The "–+" library manager command indicates to WLIB that the object file should be replaced in the library.

The file "WINDOW.MIF" contains the makefile declarations that are common to every memory model. The ".MIF" extension will be used for all the Make Include Files discussed in this manual. This extension is also in the default extension list so it is a recommended extension for Make include files. The contents of the "WINDOW.MIF" file is as follows:

```
#
# example of a Make Include File
#
common = /d1 /w3            # common options
objs = window.obj bios.obj keyboard.obj mouse.obj

.c: ..\c
.c.obj:
        wcc386 $[* $(common) $(local) /m$(model)

window_$(model).lib : $(objs)
        wlib window_$(model) @..\window
```

The macros "model" and "local" are defined by the file "MAKEFILE" in each object directory. An example of the file "MAKEFILE" in the medium memory model object directory is:

```
#
# !include example
#
model = m           # memory model required
local =             # memory model specific options
!include ..\window.mif
```

Notice that changes that affect all the memory models may be made in one file, namely "WINDOW.MIF". Any changes that are specific to a memory model may be made to the "MAKEFILE" in the object directory. To update the medium memory model library, the following commands may be executed:

Example:

```
C>cd \window\bcsd
C>wmake
```

A DOS ".BAT" or OS/2 ".CMD" file may be used to update all the different memory models. If the following DOS "MAKEALL.BAT" (OS/2 "MAKEALL.CMD") file is located somewhere in the DOS or OS/2 "PATH", we may update all the libraries.

```
cd \window\scsd
wmake %1 %2 %3 %4 %5 %6 %7 %8 %9
cd \window\scbd
wmake %1 %2 %3 %4 %5 %6 %7 %8 %9
cd \window\bcsd
wmake %1 %2 %3 %4 %5 %6 %7 %8 %9
cd \window\bcbd
wmake %1 %2 %3 %4 %5 %6 %7 %8 %9
cd \window\bchd
wmake %1 %2 %3 %4 %5 %6 %7 %8 %9
```

The batch file parameters are useful if you want to specify options to Make. For instance, a global recompile may be done by executing:

Example:

```
C>makeall /a
```

The `!include` preprocessing directive is a good way to partition common information so that it may be maintained easily.

Another use of the `!include` involves program generated makefile information. For instance, if we have a program called "WMKMK" that will search through source files and generate a file called "WMKMK.MIF" that contains:

```
#
# program generated makefile information
#
C_to_OBJ = wcc386 $[* $(wcc_options)

OBJECTS = WINDOW.OBJ BIOS.OBJ KEYBOARD.OBJ MOUSE.OBJ

WINDOW.OBJ : ..\C\WINDOW.C ..\H\PROTO.H ..\H\GLOBALS.H
    $(C_to_OBJ)
BIOS.OBJ : ..\C\BIOS.C ..\H\BIOS_DEF.H ..\H\GLOBALS.H
    $(C_to_OBJ)
KEYBOARD.OBJ : ..\C\KEYBOARD.C ..\H\PROTO.H ..\H\GLOBALS.H
    $(C_to_OBJ)
MOUSE.OBJ : ..\C\MOUSE.C ..\H\PROTO.H ..\H\GLOBALS.H
    $(C_to_OBJ)
```

In order to use this program generated makefile information, we use a "MAKEFILE" containing:

```
#
# makefile that makes use of generated makefile information
#
wcc_options = /mf /d1 /w3

first_target : window.lib .SYMBOLIC
        echo done

!include wmkmk.mif

window.lib : $(OBJECTS)
        wlib window $(OBJECTS)

make : .SYMBOLIC
        wmkmk /r ..\c\*.c+..\h
```

Notice that there is a symbolic target "first_target" that is used as a "place holder". The default behaviour for Make is to "make" the first target encountered in the makefile. The symbolic target "first_target" ensures that we have control over what file will be updated first (in this case "WINDOW.LIB"). The use of the `!include` preprocessing directive simplifies the use of program generated makefile information because any changes are localized to the file "MAKEFILE". As program development continues, the file "WMKMK.MIF" may be regenerated so that subsequent invocations of WMAKE benefit from the new makefile information. The file "MAKEFILE" even contains the command to regenerate the file "WMKMK.MIF". The symbolic target "make" has an associated command list that will regenerate the file "WMKMK.MIF". The command list can be executed by typing the following command:

Example:

```
C>wmake make
```

The use of the `!include` preprocessing directive is a simple way to reduce maintenance of related makefiles.

Hint: Macros are expanded on `!include` preprocessor control lines. This allows many benefits like:

!include $(%env_var)

so that the files that Make will process can be controlled through many different avenues like internal macros, command line macros, and environment variables.

Another way to access files is through the suffix path feature of Make. A definition like

.mif : c:\mymifs;d:\some\more\mifs

will cause Make to search different paths for any make include files.

7.16.2 Conditional Processing

Make has conditional preprocessing directives available that allow different declarations to be processed. The conditional preprocessing directives allow the makefile to

1. check whether a macro is defined, and
2. check whether a macro has a certain value.

The macros that can be checked include

1. normal macros "$(<macro_identifier>)"
2. environment macros "$(%<environment_var>)"

The conditional preprocessing directives allow a makefile to adapt to different external conditions based on the values of macros or environment variables. We can define macros on the WMAKE command line as shown in the following example.

Example:

```
C>wmake "macro=some text with spaces in it"
```

Alternatively, we can include a makefile that defines the macros if all the macros cannot fit on the command line. This is shown in the following example:

Example:

```
C>wmake /f macdef.mif /f makefile
```

Also, environment variables can be set before WMAKE is invoked. This is shown in the following example:

Example:

```
C>set macro=some text with spaces in it
C>wmake
```

Now that we know how to convey information to Make through either macros or environment variables, we will look at how this information can be used to influence makefile processing.

Make has conditional preprocessing directives that are similar to the C preprocessor directives. Make supports these preprocessor directives:

```
!ifeq
!ifneq
!ifdef
!ifndef
```

along with

```
!else
!endif
```

Together these preprocessor directives allow selection of makefile declarations to be based on either the value or the existence of a macro.

Environment variables can be checked by using an environment variable name prefixed with a "%". A common use of a conditional preprocessing directive involves setting environment variables.

```
#
# setting an environment variable
#
!ifndef %lib

.BEFORE
        set lib=c:\penpoint\sdk\lib;c:\penpoint\sdk\lib\dos
!endif
```

If you are writing portable applications, you might want to have:

```
#
# checking a macro
#
!include version.mif

!ifdef OS2
machine = /2            # compile for 80286
!else
machine = /0            # default: 8086
!endif
```

The `!ifdef` ("if defined") and `!ifndef` ("if not defined") conditional preprocessing directives are useful for checking boolean conditions. In other words, the `!ifdef` and `!ifndef` are useful for "yes-no" conditions. There are instances where it would be useful to check a macro against a value. In order to use the value checking preprocessor directives, we must know the exact value of a macro. A macro definition is of the form:

```
<macro_identifier> = <text> <comment>
```

Make will first strip any comment off the line. The macro definition will then be the text following the equal "=" sign with leading and trailing blanks removed. Initially this might not seem like a sensible way to define a macro but it does lend itself well to defining macros that are common in makefiles. For instance, it allows definitions like:

```
#
# sample macro definitions
#
wcc_options   = /d1 /s        # line numbers, no stack checking
wlink_options = debug line    # line number debugging
```

These definitions are both readable and useful. A makefile can handle differences between compilers with the `!ifeq` and `!ifneq` conditional preprocessing directives. One way of setting up adaptive makefiles is:

```
#
# options made simple
#
compiler          = wcc386

stack_overflow    = no    # yes -> check for stack overflow
line_info         = yes   # yes -> generate line numbers
```

```
!ifeq compiler wcc386
!ifneq stack_overflow    yes
stack_option     =       /s
!endif
!ifeq line_info          yes
line_option      =       /d1
!endif
!endif

!ifeq compiler tcc
!ifeq stack_overflow     yes
stack_option     =       -N
!endif
!ifeq line_info          yes
line_option      =       -y
!endif
!endif

#
# make sure the macros are defined
#
!ifndef stack_option
stack_option     =
!endif
!ifndef line_option
line_option      =
!endif

example : .SYMBOLIC
        echo $(compiler) $(stack_option) $(line_option)
```

The conditional preprocessing directives can be very useful to hide differences, exploit similarities, and organize declarations for applications that use many different programs.

Another directive is the `!define` directive. This directive is equivalent to the normal type of macro definition (i.e., macro = text) but will make C programmers feel more at home. One important distinction is that the `!define` preprocessor directive may be used to reflect the logical structure of macro definitions in conditional processing. For instance, the previous makefile could have been written in this style:

```
!ifndef stack_option
!  define stack_option
!endif
!ifndef line_option
!  define line_option
!endif
```

The "!" character must be in the first column but the directive keyword can be indented. This freedom applies to all of the preprocessing directives. The `!else` preprocessing directive benefits from this type of style because `!else` can also check conditions like:

```
!else ifeq
!else ifneq
!else ifdef
!else ifndef
```

so that logical structures like:

```
!ifdef %version
!  ifeq %version debugging
!    define option = debug all
!  else ifeq %version beta
!    define option = debug line
!  else ifeq %version production
!    define option = debug
!  else
!    error invalid value in VERSION
!  endif
!endif
```

can be used. The above example checks the environment variable "VERSION" for three possible values and acts accordingly.

Another derivative from the C language preprocessor is the `!error` directive which has the form of

```
!error <text>
```

in Make. This directive will print out the text and terminate processing of the makefile. It is very useful in preventing errors from macros that are not defined properly. Here is an example of the `!error` preprocessing directive.

```
!ifndef stack_option
!  error stack_option is not defined
!endif
!ifndef line_option
!  error line_option is not defined
!endif
```

There is one more directive that can be used in a makefile. The `!undef` preprocessing directive will clear a macro definition. The `!undef` preprocessing directive has the form:

```
!undef <macro_identifier>
```

The macro identifier can represent a normal macro or an environment variable. A macro can be cleared after it is no longer needed. Clearing a macro will reduce the memory requirements for a makefile. If the macro identifier represents an environment variable (i.e., the identifier has a "%" prefix) then the environment variable will be deleted from the current environment. The `!undef` preprocessing directive is useful for deleting environment variables and reducing the amount of internal memory required during makefile processing.

7.17 Command List Directives

Make supports special directives that provide command lists for different purposes. If a command list cannot be found while updating a target then the directive `.DEFAULT` may be used to provide one. A simple `.DEFAULT` command list which makes the target appear to be updated is:

```
.DEFAULT
        wtouch $^@
```

The WATCOM Touch utility sets the time-stamp on the file to the current time. The effect of the above rule will be to "update" the file without altering its contents.

In some applications it is necessary to execute some commands before any other commands are executed and likewise it is useful to be able to execute some commands after all other commands are executed. Make supports this capability by checking to see if the `.BEFORE` and `.AFTER` directives have been used. If the `.BEFORE` directive has been used, the `.BEFORE` command list is executed before any commands are executed. Similarly the `.AFTER` command list is executed after processing is finished. It is important to note that if all the files are up to date and no commands must be executed, the `.BEFORE` and `.AFTER` command lists are never executed. If some commands are executed to update targets and errors are detected (non-zero return status, macro expansion errors), the `.AFTER` command list is not executed (the `.ERROR` directive supplies a command list for error conditions and is discussed in this section). These two directives may be used for maintenance as illustrated in the following example:

```
#
# .BEFORE and .AFTER example
#
.BEFORE
        echo .BEFORE command list executed
.AFTER
        echo .AFTER command list executed
#
# rest of makefile follows
#
        .
        .
        .
```

If all the targets in the makefile are up to date then neither the `.BEFORE` nor the `.AFTER` command lists will be executed. If any of the targets are not up to date then before any commands to update the target are executed, the `.BEFORE` command list will be executed. The `.AFTER` command list will be executed only if there were no errors detected during the updating of the targets. The `.BEFORE`, `.DEFAULT`, and `.AFTER` command list directives provide the capability to execute commands before, during, and after the makefile processing.

Make also supports the `.ERROR` directive. The `.ERROR` directive supplies a command list to be executed if an error occurs during the updating of a target.

```
#
# .ERROR example
#
.ERROR
        beep
#
# rest of makefile follows
#
        .
        .
        .
```

The above makefile will audibly signal you that an error has occurred during the makefile processing. If any errors occur during the `.ERROR` command list execution, makefile processing is terminated.

7.18 MAKEINIT File

As you become proficient at using Make, you will probably want to isolate common makefile declarations so that there is less duplication among different makefiles. Make will search for a file called "MAKEINIT" and process it before any other makefiles. The search for the "MAKEINIT" file will occur along the current DOS or OS/2 "PATH". If the file "MAKEINIT" is not found, processing continues without any errors. The only default declaration that Make provides is equivalent to a "MAKEINIT" file containing:

```
__MAKEOPTS__ = <options passed to WMAKE>
__MAKEFILES__ = <list of makefiles>
__MSDOS__ =
# clear .EXTENSIONS list
.EXTENSIONS:

# set .EXTENSIONS list
.EXTENSIONS: .exe .exp .lib .obj .asm .c .for .pas .cob .h .fi .mif
```

For OS/2, the `__MSDOS__` macro will be replaced by `__OS2__`. The use of a "MAKEINIT" file will allow you to reuse common declarations and will result in simpler, more maintainable makefiles.

7.19 Command List Execution

Make is a program which must execute other programs and DOS or OS/2 commands. There are three basic types of executable files in DOS.

1. `.COM` files
2. `.EXE` files
3. `.BAT` files

There are two basic types of executable files in OS/2.

1. `.EXE` files
2. `.CMD` files

The `.COM` and `.EXE` files may be loaded into memory and executed. The `.BAT` files must be executed by the DOS command processor or shell, "COMMAND.COM". The `.CMD` files must be executed by the OS/2 command processor or shell, "CMD.EXE" Make will search along the DOS or OS/2 "PATH" for the command and depending on the file extension the file will be executed in the proper manner. If Make detects any input or output redirection characters (these are ">", "<", and "|") in the command, it will be executed by the shell. The exclamation mark prefix (!) will force a command to be executed by the shell. Also, the

command will be executed by the shell if the command is an internal shell command from the following list:

break	(check for Ctrl+Break)
call	(nest batch files)
chdir	(change current directory)
cd	(change current directory)
cls	(clear the screen)
cmd	(start OS/2 command processor)
command	(start DOS command processor)
copy	(copy or combine files)
ctty	(DOS redirect input/output to COM port)
d:	(change drive where "d" represents a drive specifier)
date	(set system date)
del	(erase files)
dir	(display contents in a directory)
echo	(display commands as they are processed)
erase	(erase files)
for	(repetitively process commands, intercepted by WMAKE)
if	(allow conditional processing of commands)
md	(make directory)
mkdir	(make directory)
path	(set search path)
pause	(suspend batch operations)
prompt	(change command prompt)
ren	(rename files)
rename	(rename files)
rmdir	(remove directory)
rd	(remove directory)
set	(set environment variables, intercepted by WMAKE)
time	(set system time)
type	(display contents of a file)
ver	(display the DOS or OS/2 version number)
verify	(set data verification)
vol	(display disk volume label)

The DOS or OS/2 "SET" command is intercepted by Make. The "SET" command may be used to set environment variables to values required during makefile processing. The environment variable changes are only valid during makefile processing and do not affect the values that were in effect before Make was invoked. The "SET" command may be used to initialize environment variables necessary for the makefile commands to execute properly. The setting of environment variables in makefiles reduces the number of "SET" commands required in the DOS or OS/2 system initialization file. Here is an example with the WATCOM C/386 compiler.

```
#
# SET example
#
.BEFORE
        set include=c:\special\h;$(%include)
        set lib=c:\penpoint\sdk\lib;c:\penpoint\sdk\lib\dos
#
# rest of makefile follows
#
        .
        .
        .
```

The first "SET" command will set up the **INCLUDE** environment variable so that the WATCOM C/386 compiler may find header files. Notice that the old value of the **INCLUDE** environment variable is used in setting the new value.

The second "SET" command indicates to the WATCOM Linker that libraries may be found in the indicated directories.

Environment variables may be used also as dynamic variables that may communicate information between different parts of the makefile. An example of communication within a makefile is illustrated in the following example.

```
#
# internal makefile communication
#
.BEFORE
        set message=message text 1
        echo *$(%message)*
        set message=
        echo *$(%message)*

.example : another_target .SYMBOLIC
        echo *$(%message)*

another_target : .SYMBOLIC
        set message=message text 2
```

The output of the previous makefile would be:

```
(command output only)
*message text 1*
**
*message text 2*
```

Make handles the "SET" command so that it appears to work in an intuitive manner similar to the DOS or OS/2 "SET" command. The "SET" command also may be used to allow commands to relay information to commands that are executed afterwards.

The DOS or OS/2 "FOR" command is intercepted by Make. The reason for this is that DOS has a fixed limit for the size of a command thus making it unusable for large makefile applications. One such application that can be done easily with Make is the construction of a WLINK command file from a makefile. The idea behind the next example is to have one file that contains the list of object files. Anytime this file is changed, say, after a new module has been added, a new linker command file will be generated which in turn, will cause the linker to relink the executable. First we need the makefile to define the list of object files, this file is "OBJDEF.MIF" and it declares a macro "objs" which has as its value the list of object files in the application. The content of the "OBJDEF.MIF" file is:

```
#
# list of object files
#
objs = &
     window.obj &
     bios.obj &
     keyboard.obj &
     mouse.obj
```

The main makefile ("MAKEFILE") is:

```
#
# FOR command example
#
!include objdef.mif

plot.exp : $(objs) plot.lnk
        wlink @plot

plot.lnk : objdef.mif
        echo NAME $^& >$^@
        echo DEBUG all >>$^@
        for %i in ($(objs)) do echo FILE %i >>$^@
```

This makefile would produce a file "PLOT.LNK" automatically whenever the list of object files is changed (anytime "OBJDEF.MIF" is changed). For the above example, the file "PLOT.LNK" would contain:

```
NAME plot
DEBUG all
FILE window.obj
FILE bios.obj
FILE keyboard.obj
FILE mouse.obj
```

Make supports eight internal commands:

1. `%null`
2. `%stop`
3. `%quit`
4. `%abort`
5. `%create`
6. `%write`
7. `%append`
8. `%make`

The `%null` internal command does absolutely nothing. It is useful because Make demands that a command list be present whenever a target is updated.

```
#
# %null example
#
all : application1 application2 .SYMBOLIC
        %null

application1 : appl1.exp .SYMBOLIC
        %null

application2 : appl2.exp .SYMBOLIC
        %null

appl1.exp : (dependents ...)
        (commands)

appl2.exp : (dependents ...)
        (commands)
```

Through the use of the `%null` internal command, multiple application makefiles may be produced that are quite readable and maintainable. The `%stop` internal command will temporarily suspend makefile processing and print out a message asking whether the Makefile processing should continue. Make will wait for either the "y" key (indicating that the Makefile processing should continue) or the "n" key. If the "n" key is pressed, makefile processing will stop. The `%stop` internal command is very useful for debugging makefiles

but it may be used also to develop interactive makefiles. The %quit internal command will terminate execution of Make and return to DOS or OS/2 with an exit code of zero. The %abort internal command is identical to %quit except that a non-zero exit code is returned by WMAKE.

The %create, %write, and %append internal commands allow WMAKE to generate files under makefile control. This is useful for files that have contents that depend on makefile contents. Through the use of macros and the "for" command, Make becomes a very powerful tool in maintaining lists of files for other programs. The %create internal command will create or truncate a file so that the file does not contain any text. The %create internal command has the form:

```
%create <file>
```

where <file> is a file specification. The %write internal command will create or truncate a file and write one line of text into it. The %append internal command will append a text line to the end of a file (which will be created if it does not exist). The %write and %append internal commands have the same form, namely:

```
%write <file> <text>
%append <file> <text>
```

where <file> is a file specification and <text> is arbitrary text. Full macro processing is performed on these internal commands so the full power of WMAKE can be used. The following example illustrates a common use of these internal commands.

```
#
# %create %append example
#
!include objdef.mif

plot.exp : $(objs) plot.lnk
        wlink @plot

plot.lnk : objdef.mif
        %create $^@
        %append $^@ NAME $^&
        %append $^@ DEBUG all
        for %i in ($(objs)) do %append $^@ FILE %i
```

The above code demonstrates a valuable technique that can generate directive files for WLINK, WLIB, and other utilities.

The %make internal command permits the updating of a specific target. The %make internal command has the form:

```
%make <target>
```

where <target> is a target in the makefile.

```
#
# %make example
#
!include objdef.mif

plot.exp : $(objs)
        %make plot.lnk
        wlink @plot

plot.lnk : objdef.mif
        %create $^@
        %append $^@ NAME $^&
        %append $^@ DEBUG all
        for %i in ($(objs)) do %append $^@ FILE %i
```

7.20 Compatibility Between WATCOM Make and UNIX Make

Make was originally based on the UNIX Make utility. The DOS or OS/2 environment presents a base of users which may or may not be familiar with the UNIX operating system. Make is designed to be a DOS or OS/2 product with some UNIX compatibility. The line continuation in UNIX Make is a backslash ("\") at the end of the line. The backslash ("\") is used by DOS or OS/2 for directory specifications and as such will be confused with line continuation. For example, you could type:

```
cd \
```

along with other commands ... and get unexpected results. However, if your makefile doesn't contain "\" path separator characters and you wish to use "\" as a line continuation indicator then you can use the Make "u" (UNIX compatibility mode) option.

Also, in the UNIX operating system there is no concept of file extensions, only the concept of a file suffix. Make will accept the UNIX Make directive `.SUFFIXES` for compatibility with UNIX makefiles. The UNIX compatible special macros supported are:

Macro	*Expansion*
$@	full name of the target
$*	target with the extension removed
$<	list of all dependents
$?	list of dependents that are younger than the target

The extra checking of makefiles done by Make will require modifications to UNIX makefiles. The UNIX Make utility does not check for the existence of targets after the associated command list is executed so the "c" or the `.NOCHECK` directive should be used to disable this checking. The lack of a command list to update a target is ignored by the UNIX Make utility but WATCOM Make requires the special internal command `%null` to specify a null command list. In summary, Make supports many of the features of the UNIX Make utility but is not 100% compatible.

8 The Touch Utility

This chapter describes the WATCOM Touch utility. WATCOM Touch will set the time-stamp (i.e., the modification date and time) of one or more files. The new modification date and time may be the current date and time, the modification date and time of another file, or a date and time specified on the command line. This utility is normally used in conjunction with the WATCOM Make utility. The rationale for bringing a file up-to-date without altering its contents is best understood by reading the chapter which describes the Make utility.

The WATCOM Touch command line syntax is:

WTOUCH [options] file_spec [file_spec...]

The square brackets [ ] denote items which are optional.

options is a list of valid options, each preceded by a slash ("/") or a dash ("–"). Options may be specified in any order.

/c	do not create an empty file if the specified file does not exist
/d<date>	specify the date for the file time-stamp
/f<file>	use the time-stamp from the specified file
/t<time>	specify the time for the file time-stamp

file_spec is the DOS or OS/2 file specification for the file to be touched. Any number of file specifications may be listed. The DOS or OS/2 wild card characters "*" and "?" may be used.

8.1 WTOUCH Operation

WATCOM WTOUCH is used to set the time-stamp (i.e., the modification date and time) of a file. The contents of the file are not affected by this operation. If the specified file does not exist, it will be created as an empty file. This behaviour may be altered with the "c" option so that if the file is not present, a new empty file will not be created.

Example:

```
(will not create myfile.dat)
C>wtouch /c myfile.dat
```

If a wild card file specification is used and no files match the pattern, no files will have their time-stamps altered. The date and time that all the specified files are set to is determined as follows:

1. The current date and time is used as a default value.

2. A time-stamp from an "age file" may replace the current date and time. The "f" option is used to specify the file that will supply the time-stamp.

   Example:

   ```
   (use the date and time from file "last.tim")
   C>wtouch /f last.tim file*.dat
   ```

3. The date and/or time may be specified from the command line to override a part of the time-stamp that will be used. The "d" and "t" options are used to override the date and time respectively.

   Example:

   ```
   (use current date but use different time)
   C>wtouch /t 2:00p file*.dat
   (completely specify date and time)
   C>wtouch /d 10-31-90 /t 8:00:00 file*.dat
   (use date from file "last.tim" but set time)
   C>wtouch /f last.tim /t 12:00 file*.dat
   ```

   The format of the date and time on the command line depends on the country information provided by the host operating system. WATCOM Touch should accept dates and times in a similar format to any operating system utilities (i.e., the DATE and TIME utilities provided by DOS). The "a" and "p" suffix is an extension to the time syntax for specifying whether the time is A.M. or P.M., but this is only available if the operating system is not configured for military or 24-hour time.

9 The Object File Disassembler

9.1 Introduction

This chapter describes the WATCOM Disassembler. It takes as input an object file (a file with extension "obj") and produces, as output, the Intel assembly language equivalent. The WATCOM C/386 compiler does not produce an assembly language listing of a C program. Instead, the WATCOM Disassembler can be used to generate an assembly language listing from the object file generated by WATCOM C/386.

The WATCOM Disassembler command line syntax is the following.

WDISASM [options] [d:][path]filename[.ext] [options]

The square brackets [ ] denote items which are optional.

WDISASM is the name of the WATCOM Disassembler.

d: is an optional drive specification such as "A:", "B:", etc. If not specified, the default drive is assumed.

path is an optional path specification such as "\PROGRAMS\OBJ\". If not specified, the current directory is assumed.

filename is the DOS or OS/2 file name of the object file to disassemble.

ext is the DOS or OS/2 file extension of the object file to disassemble. If omitted, a file extension of "obj" is assumed. If the period "." is specified but not the extension, the file is assumed to have no file extension.

options is a list of valid options, each preceded by a slash ("/") or a dash ("–"). Options may be specified in any order.

/l[=<list_file>]
: create a listing file

/s[=<source_file>]
: using object file source line information, imbed original source lines into the output file

/a
: write assembly instructions only to the listing file

/au
: write assembly instructions only to the listing file using UNIX-style assembly syntax

/b
: generate alternate form of based or indexed addressing modes

/e
: include list of external names

/u
: display instruction opcode mnemonics in upper case

/p
: include list of public names

/r
: display register names in upper case

/c=<code_name>
: name of additional segments that contain executable code

The following sections describe the list of options.

9.2 The Listing Option - "l[=<list_file>]"

By default, the WATCOM Disassembler produces its output to the terminal. The "l" (lowercase L) option instructs the WATCOM Disassembler to produce the output to a listing file. The default file name of the listing file is the same as the file name of the object file. The default file extension of the listing file is .lst.

Example:

```
C>wdisasm calendar /l
```

In the above example, the WATCOM Disassembler is instructed to disassemble the contents of the file `calendarobj` and produce the output to a listing file called `calendar.lst`.

An alternate form of this option is "l=<list_file>". With this form, you can specify the name of the listing file. When specifying a listing file, a file extension of ".lst" is assumed if none is specified.

Example:

```
C>wdisasm calendar /l=calendar.lis
```

In the above example, the WATCOM Disassembler is instructed to disassemble the contents of the file `calendarobj` and produce the output to a listing file called `calendar.lis`.

9.3 The Source Option - "s[=<source_file>]"

The "s" option causes the source lines corresponding to the assembly language instructions to be produced in the listing file. The object file must contain line numbering information. That is, the "d1" or "d2" option must have been specified when the source file was compiled. If no line numbering information is present in the object file, the "s" option is ignored.

The following defines the order in which the source file name is determined when the "s" option is specified.

1. If present, the source file name specified on the command line.
2. The name from the module header record.
3. The object file name.

Example:

```
C>wcc386 mysrc /d1
C>wdisasm mysrc /s /l
```

In the above example, the WATCOM Disassembler is instructed to disassemble the contents of the file `mysrcobj` and produce the output to the listing file `mysrc.lst`. The source lines are extracted from the file `mysrc.c`.

An alternate form of this option is "s=<source_file>". With this form, you can specify the name of the source file.

Example:

```
C>wdisasm mysrc /s=myprog.c /l
```

The above example produces the same result as in the previous example except the source lines are extracted from the file `myprog.c`.

9.4 The Assembly Format Option - "a", "au"

The "a" option controls the format of the output produced to the listing file. When specified, the WATCOM Disassembler will produce a listing file that can be used as input to an assembler. When "au" is specified, the assembly syntax is that employed by UNIX systems.

Example:

```
C>wdisasm calendar /a /l=calendar.asm
```

In the above example, the WATCOM Disassembler is instructed to disassemble the contents of the file `calendarobj` and produce the output to the file `calendar.asm` so that it can be assembled by an assembler.

9.5 The Uppercase Opcode Option - "u"

The "u" option instructs the WATCOM Disassembler to display instruction opcode mnemonics in uppercase. The default is to display instruction opcode mnemonics in lowercase.

9.6 The Uppercase Register Option - "r"

The "r" option instructs the WATCOM Disassembler to display register names in uppercase. The default is to display register names in lowercase.

9.7 The Alternate Addressing Form Option - "b"

The "b" option causes an alternate form of the based or indexed addressing mode to be used in an instruction. By default, the following form is used.

```
mov ax,-2[bp]
```

If the "b" option is specified, the following form is used.

```
mov ax,[bp-2]
```

9.8 The External Symbols Option - "e"

The "e" option controls the amount of information produced in the listing file. When specified, a list of all externally defined symbols is produced in the listing file.

Example:

```
C>wdisasm calendar /e
```

In the above example, the WATCOM Disassembler is instructed to disassemble the contents of the file `calendarobj` and produce the output, with a list of all external symbols, on the screen. A sample list of external symbols is shown below.

```
List of external symbols

Symbol
----------------
__STK            00000349 0000030b 000002bc 00000204 000000ca 00000007
int386_          00000378 0000033b
localtime_       00000028
memset_          000002d9
printf_          000002f8 000001e2 000001d0 0000014a 00000135
strlen_          00000108
time_            0000001d
------------------------------------------------------------
```

Each externally defined symbol is followed by a list of location counter values indicating where the symbol is referenced.

The "e" option is ignored when the "a" option is specified.

9.9 The Public Symbols Option - "p"

The "p" option controls the amount of information produced in the listing file. When specified, a list of all public symbols is produced in the listing file.

Example:

```
C>wdisasm calendar /p
```

In the above example, the WATCOM Disassembler is instructed to disassemble the contents of the file `calendarobj` and produce the output, with a list of all exported symbols, to the screen. A sample list of public symbols is shown below.

```
List of public symbols

SYMBOL          GROUP           SEGMENT         ADDRESS
-------------------------------------------------------
__main_entry_   DGROUP          CONST           00000000
Box_                            _TEXT           000001fd
Calendar_                       _TEXT           000000c3
ClearScreen_                    _TEXT           00000342
Line_                           _TEXT           000002b5
main_                           _TEXT           00000000
PosCursor_                      _TEXT           00000304

-------------------------------------------------------
```

The "p" option is ignored when the "a" option is specified.

9.10 Identifying Code Segments Option - "c=<code_name>"

The "c" option permits you to specify one additional segment name or pattern for segments that contain executable code. Valid forms of the "c" option are:

```
c=<string>
c=*<string>
c=<string>*
```

`<string>` is any sequence of characters with letters treated in a case insensitive manner. The "*" character is a wildcard character.

By default, the following segments are assumed to contain executable code.

1. segments with class name "CODE"
2. segments whose name ends in "TEXT" or "CODE" (this is equivalent to "c=*text" and "c=*code")

Consider the following example.

```
C>wdisasm myprog /c=T@*
```

All segments whose name starts with the characters "T@" (in addition to those ending in "CODE" or "TEXT") are assumed to contain executable code.

> ***Note:*** It is not possible to instruct the WATCOM Disassembler to interpret segments whose name ends in "CODE" or "TEXT" as data.

9.11 An Example

Consider the following WATCOM C/386 program contained in the file `hello.c`.

```
#include <stdio.h>

void main()
    {
        printf( "Hello world\n" );
    }
```

We can compile it by issuing the following command.

```
C>wcc386 hello /d1
```

An object file called `helloobj` will be produced. The "d1" option causes line numbering information to be generated in the object file. We can use the WATCOM Disassembler to disassemble the contents of the object file by issuing the following command.

```
C>wdisasm hello /l /e /p /s /r
```

The output will be created to a listing file called `hello.lst` (the "l" option was specified"). It will contain a list of external symbols (the "e" option was specified), a list of public symbols (the "p" option was specified) and the source lines corresponding to the assembly language instructions (the "s" option was specified). The source input file is called `hello.c`. The register names will be displayed in upper case (the "r" option was specified). The contents of `hello.lst` are as follows.

```
Module: hello.c
Group: 'DGROUP' CONST,_DATA,_BSS

Segment: '_TEXT' BYTE USE32  0000001a bytes

#include <stdio.h>

void main()
  {
 0000  50                  main_          push    EAX
 0001  b8 08 00 00 00                     mov     EAX,00000008H
 0006  e8 00 00 00 00                     call    __STK
 000b  58                                 pop     EAX

    printf( "Hello world\n" );
 000c  68 04 00 00 00                     push    offset L1
 0011  e8 00 00 00 00                     call    printf_
 0016  83 c4 04                           add     ESP,0004H

  }
 0019  c3                                 ret
No disassembly errors

List of external symbols

Symbol
----------------
__STK            00000007
printf_          00000012
------------------------------------------------------------

Segment: 'CONST' WORD USE32  00000011 bytes
 0000  00 00 00 00              __main_entry_   DD      main_
 0004  48 65 6c 6c 6f 20 77 6f L1               - Hello wo
 000c  72 6c 64 0a 00                           - rld..
No disassembly errors

------------------------------------------------------------
List of public symbols

SYMBOL           GROUP           SEGMENT          ADDRESS
------------------------------------------------------------
__main_entry_    DGROUP          CONST            00000000
main_                            _TEXT            00000000

------------------------------------------------------------
```

Let us create a form of the listing file that can be used as input to an assembler.

```
C>wdisasm hello /l=hello.asm /r /a
```

The output will be produced in the file `hello.asm`. Its contents are as follows.

```
                NAME     hello
                EXTRN    _cstart_:WORD
                EXTRN    printf_:WORD
                EXTRN    __STK:WORD
DGROUP          GROUP    CONST,_DATA,_BSS
_TEXT           SEGMENT  PUBLIC BYTE USE32 'CODE'
                ASSUME   CS:_TEXT,DS:DGROUP
                PUBLIC   main_
main_:          push     eax
                mov      eax,00000008H
                call     near ptr __STK
                pop      eax
                push     offset DGROUP:L1
                call     near ptr printf_
                add      esp,0004H
                ret
_TEXT           ENDS
CONST           SEGMENT  PUBLIC WORD USE32 'DATA'
                PUBLIC   __main_entry_
__main_entry_   LABEL    BYTE
                DD       main_
L1              LABEL    BYTE
                DB       48H,65H,6cH,6cH,6fH,20H,77H,6fH
                DB       72H,6cH,64H,0aH,00H
CONST           ENDS

_DATA           SEGMENT  PUBLIC WORD USE32 'DATA'
_DATA           ENDS

_BSS            SEGMENT  PUBLIC WORD USE32 'BSS'
_BSS            ENDS

                END
```

10 The WATCOM Patch Utility

The WATCOM Patch Utility is a utility program which may be used to apply patches or bug fixes to WATCOM C/386 and its associated tools. As problems are reported and fixed, registered users of WATCOM C/386 will be notified of patches that they can apply to resolve these problems.

10.1 Applying a Patch

The format of the BPATCH command line is:

BPATCH [options] patch_file

The square brackets [ ] denote items which are optional.

where:	***description:***
options	is a list of valid WATCOM Patch Utility options, each preceded by a dash ("–"). Options may be specified in any order. The possible options are:
	-p Do not prompt for confirmation
	-b Do not create a .BAK file
	-q Print current patch level of file
patch_file	is the file specification for a patch file provided by WATCOM.

Suppose a patch file called "wcc386.a" is supplied by WATCOM to fix a bug in the file "WCC386.EXE". The patch may be applied by typing the command:

```
bpatch wcc386.a
```

The WATCOM Patch Utility locates the file `C:\PENPOINT\SDK\BIN\WCC386.EXE` using the **PATH** environment variable. The actual name of the executable file is extracted

from the file `wcc386.a`. It then verifies that the file to be patched is the correct one by comparing the size of the file to be patched to the expected size. If the file sizes match, the program responds with:

```
Ok to modify 'C:\PENPOINT\SDK\BIN\WCC386.EXE'? [y|n]
```

If you respond with "yes", BPATCH will modify the indicated file. If you respond with "no", BPATCH aborts. Once the patch has been applied the resulting file is verified. First the file size is checked to make sure it matches the expected file size. If the file size matches, a check-sum is computed and compared to the expected check-sum.

Notes:

1. If an error message is issued during the patch process, the file that you specified to be patched will remain unchanged.

2. If a sequence of patch files exist, such as "wcc386.a", "wcc386.b" and "wcc386.c", the patches must be applied in order. That is, "wcc386.a" must be applied first followed by "wcc386.b" and finally "wcc386.c".

10.2 Diagnostic Messages

If the patch cannot be successfully applied, one of the following error messages will be displayed.

Usage: %s {-p} {-q} {-b} <file>
-p = Do not prompt for confirmation
-b = Do not create a .BAK file
-q = Print current patch level of file

The command line was entered incorrectly. %s contains the name (including the full path) of the WATCOM Patch Utility.

File '%s' has not been patched

This message is issued when the "-q" option is used and the file has not been patched.

File '%s' has been patched to level '%s'

This message is issued when the "-q" option is used and the file has been patched to the indicated level.

File '%s' has already been patched to level '%s' - skipping
This message is issued when the file has already been patched to the same level or higher.

Command line may only contain one file name
More than one file name is specified on the command line.

Command line must specify a file name
No file name has been specified on the command line.

'%s' is not a WATCOM patch file
The patch file is not of the required format. The required header information is not present.

'%s' is not a valid WATCOM patch file
The patch file is not of the required format. The required header information is present but the remaining contents of the file have been corrupted.

'%s' is the wrong size (%lu1). Should be (%lu2)
The size of the file to be patched (%lu1) is not the same as the expected size (%lu2).

Cannot find '%s'
Cannot find the executable to be patched.

Cannot open '%s'
An error occurred while trying to open the patch file, the file to be patched or the resulting file.

Cannot read '%s'
An input error occurred while reading the old version of the file being patched.

Cannot rename '%s' to '%s'
The file to be patched could not be renamed to the backup file name or the resulting file could not be renamed to the name of the file that was patched.

Cannot write to '%s'
An output error occurred while writing to the new version of the file to be patched.

I/O error processing file '%s'
An error occurred while seeking in the specified file.

No memory for %s

An attempt to allocate memory dynamically failed.

Patch program aborted!

This message is issued if you answered no to the "OK to modify" prompt.

Resulting file has wrong checksum (%lu) - Should be (%lu2)

The check-sum of the resulting file (%lu) does not match the expected check-sum (%lu2). This message is issued if you have patched the wrong version.

Resulting file has wrong size (%lu1) - Should be (%lu2)

The size of the resulting file (%lu1) does not match the expected size (%lu2). This message is issued if you have patched the wrong version.

Advanced Topics

11 80386 Architecture

11.1 Registers

The register model for the 80386 microprocessor consists of 8 32-bit general-purpose registers used primarily to contain operands for arithmetic and logical operations, 6 segment registers which determine which segments of memory are addressable, and status and instruction registers used to record and alter aspects of the 80386 processor state.

11.2 General Registers

The general-purpose registers are named EAX, EBX, ECX, EDX, EBP, ESP, ESI and EDI. The low-order word of these registers have different names and can be used for handling 16-bit data items. The word registers are called AX, BX, CX, DX, BP, SP, SI and DI. Each byte of the word registers AX, BX, CX and DX has a separate name. The byte registers are named AH, AL, BH, BL, CH, CL, DH and DL and are useful for handling characters and other 8-bit data.

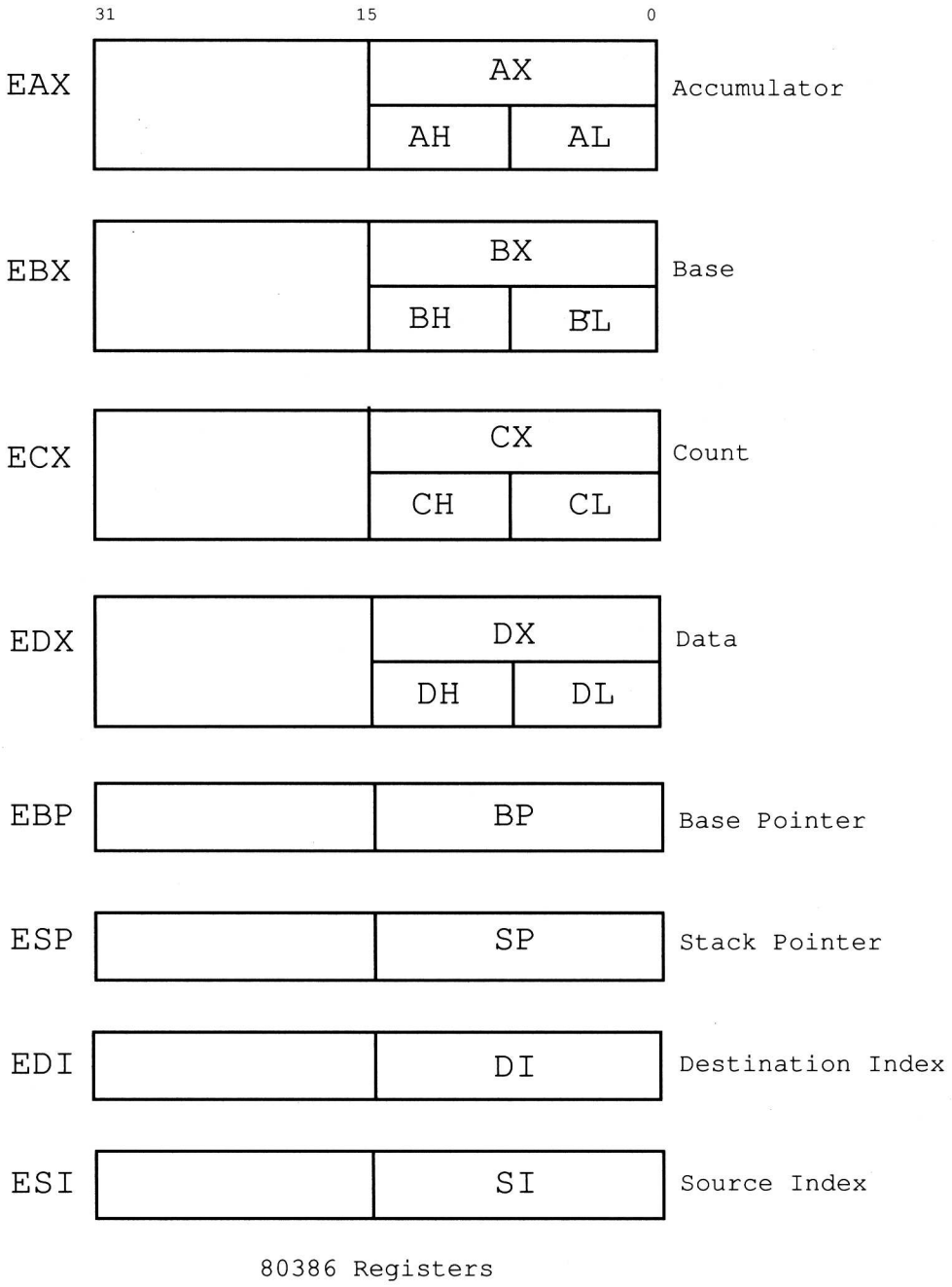


Figure 2. *80386 Registers*

All general-purpose registers can be used for addressing calculations and for results of most arithmetic and logical calculations. However, each register has specific functions as well. Instructions that use specific registers include double-precision multiply and divide, i/o, string instructions, translate, loop, variable shift and rotate, and stack operations.

11.2.1 The EAX Register

The EAX register is known as the accumulator. Some arithmetic operations implicitly use this register. For example, the multiply and divide instructions require the use of the EAX register. The input/output instructions also use this register for transferring data from input/output devices to the 80386 processor.

11.2.2 The EBX Register

The EBX register (along with the DS segment register) is used to point to the translate table for the XLAT instruction.

11.2.3 The ECX Register

The ECX register is known as the count register. In addition to being a general-purpose register, the ECX register is used as an iteration counter for the loop (LOOP, LOOPE/LOOPZ, LOOPNE/LOOPNZ, JECXZ) and the string (CMPSB/CMPSW/CMPSD, LODSB/LODSW/LODSD, MOVSB/MOVSW/MOVSD, SCASB/SCASW/SCASD, STOSB/STOSW/STOSD) instructions.

11.2.4 The EDX Register

The EDX register is known as the data register. It is used implicitly by the multiply and divide instructions. The DX register (low-order 16 bits of the EDX register) can also be used for the port address for the IN and OUT instructions.

11.2.5 The ESP and EBP Registers

These registers use the stack segment (SS) during memory address calculation. Hence they can be used to access data which resides in the stack segment. The ESP register allows the implementation of a stack in memory and hence it is referred to as the stack pointer. Instructions such as PUSH and POP operate implicitly on this register. The base pointer (EBP) can also be used to access data in the stack segment. This register is typically used to reference parameters and local variables on the stack.

11.2.6 The ESI and EDI Registers

The primary function of these registers is found in the string instructions. The data segment register (DS) in conjunction with the ESI register form the source address of the string. The extra segment register (ES) in conjunction with the EDI register form the destination address for the string move.

11.3 Segment Registers

The 6 segment registers are named CS, DS, ES, SS, FS and GS. Each segment register defines an address space of up to 4 gigabytes (4GB). Each instruction references a default segment register. In most cases, these defaults can be overridden by specifying a segment register with the operand.

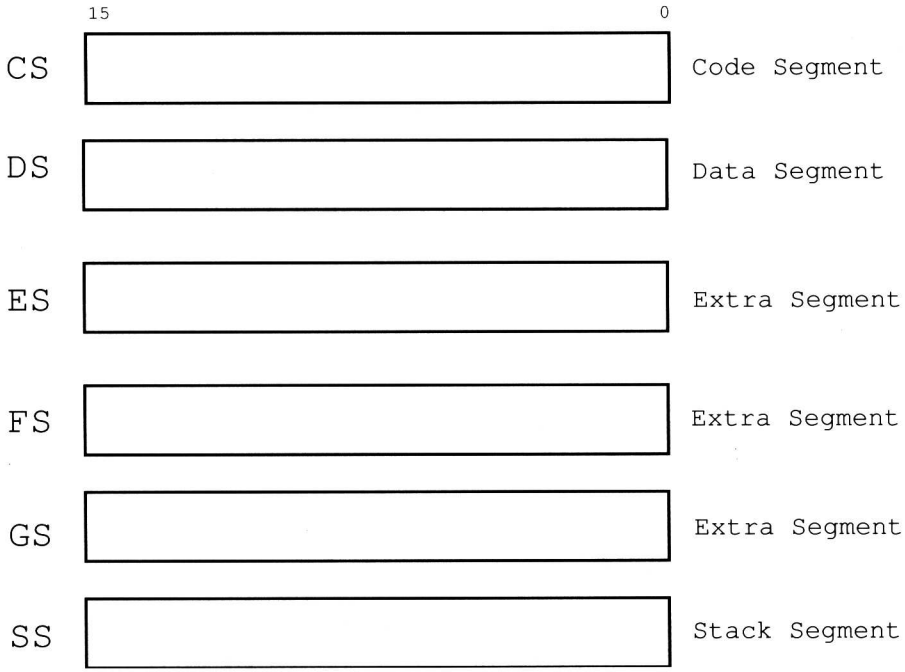


Figure 3. *Segment Registers*

11.3.1 The CS Register

The CS register is known as the code segment register. The code segment register is used in conjunction with the instruction pointer to determine the address of the instruction to be executed.

11.3.2 The DS Register

The DS register is known as the data segment register. References to data are relative to this register.

11.3.3 The ES, FS and GS Registers

The ES, FS and GS registers are additional data segment registers. The ES register is used in memory address calculations for the destination of string operations.

11.3.4 The SS Register

The SS register is known as the stack segment register. The SS register is used in memory address calculations involving the pointer registers EBP and ESP.

11.4 Instruction Pointer

The instruction pointer (EIP) is a 32-bit register which defines the offset in the address space defined by the CS segment register of the next instruction to be executed.

11.5 Flags Register

The flags register is a 32-bit register named EFLAGS which contains a number of status bits. This register is sometimes referred to as the status register. Bits in this register are numbered from 0 to 31, where 0 is the least significant bit and 31 is the most significant bit. On the 80386 only bits 0, 2, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16 and 17 are defined. The remaining bits are reserved. The low-order 16 bits of EFLAGS is called FLAGS and can be treated as a unit. This is particularly useful when executing 8086 and 80286 code since this part of EFLAGS is identical to the FLAGS register of the 8086 and 80286.

Not all instructions change the state of the flags register. You must look up the description of a particular instruction to see what effect it might have, if any, on the flags register.

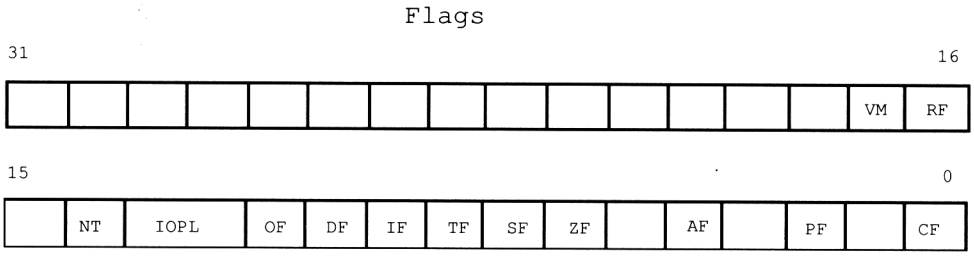


Figure 4. *Flags Register*

where:	***description:***
VM	= Virtual 8086 Mode
RF	= Resume Flag
NT	= Nested Task Flag
IOPL	= I/O Privilege Level
OF	= Overflow Flag
DF	= Direction Flag
IF	= Interrupt Flag
TF	= Trace Flag
SF	= Sign Flag
ZF	= Zero Flag
AF	= Auxiliary Carry Flag
PF	= Parity Flag
CF	= Carry Flag

of

11.5.1 The Virtual 8086 Mode

This flag, when set, indicates that the task is executing an 8086 program.

11.5.2 The Resume Flag

This flag temporarily disables debug exceptions so that an instruction can be restarted after a debug exception without immediately causing another debug exception.

11.5.3 The Nested Task Flag

The processor uses this flag to control chaining of interrupted and called tasks.

11.5.4 The I/O Privilege Level

This is used to restrict the use of input/output instructions.

11.5.5 The Carry Flag

The Carry flag is bit 0 of the status register and is referred to as CF. CF reflects a carry out of the high bit following an arithmetic operation. CF is set to 1 (TRUE) when there is a carry. CF is set to 0 (FALSE) when there is no carry.

11.5.6 The Parity Flag

The Parity flag is bit 2 of the flags register and is referred to as PF. PF is set to 1 (TRUE) when the resulting lower 8 bits of a data operation has an even number of 1 bits. An odd number of 1 bits results in the PF flag being set to 0 (FALSE).

11.5.7 The Auxiliary Carry Flag

The Auxiliary carry flag is bit 4 of the flags register and is referred to as AF. The AF flag represents a 4-bit (half-byte or nibble) carry, which is a carry out of bit 3 in an 8-bit data item. AF is set to 1 (TRUE) when there is a carry. AF is set to 0 (FALSE) when there is no carry.

11.5.8 The Zero Flag

The Zero flag is bit 6 of the flags register and is referred to as ZF. ZF is set to 1 (TRUE) when the result of the computation is zero. ZF is set to 0 (FALSE) when the result of the computation is not zero.

11.5.9 The Sign Flag

The Sign flag is bit 7 of the flags register and is referred to as SF. SF is set equal to the high-order bit of the result of a computation. SF is set to 1 (TRUE) if the result is negative. SF is set to 0 (FALSE) if the result is not negative.

11.5.10 The Trap Flag

The Trap flag is bit 8 of the flags register and is referred to as TF. When set, TF causes the 80386 microprocessor to "single step". Each instruction that is executed when TF is set results in a software interrupt being generated. This is useful in debugging when a trace of the instructions executed is desired.

11.5.11 The Interrupt Enable Flag

The Interrupt Enable flag is bit 9 of the flags register and is referred to as IF. IF determines whether interrupt processing is enabled or disabled. When IF is set to 1 (TRUE) interrupts are enabled; when IF is 0 (FALSE) interrupts are disabled. The setting of IF can be controlled using the CLI and STI instructions of the 80386.

11.5.12 The Direction Flag

The Direction flag is bit 10 of the flags register and is referred to as DF. DF determines whether string operations increment or decrement their operands. When DF is 1 (TRUE) the operands will be decremented; when DF is 0 (FALSE) the operands will be incremented. The setting of DF is controlled by the CLD and STD instructions.

11.5.13 The Overflow Flag

The Overflow flag is bit 11 of the flags register and is referred to as OF. OF represents a magnitude overflow in signed binary arithmetic. OF is set to 1 (TRUE) when the result of a 16-bit computation exceeds 32767 or the result of a 32-bit computation exceeds 2147483647 (which means that the result is now negative) or is less than -32768 in the 16-bit case or -2147483648 in the 32-bit case (which means that the result is now positive). OF is set to 0 (FALSE) when the result of the computation did not overflow.

11.6 Effective Address Calculation

Effective address calculation consists of instruction fetching and data fetching. Instruction fetching describes how flow of control is performed. Data fetching describes how data can be referenced by a program.

11.6.1 Instruction Fetching

The code segment register (CS) defines the instruction space from which instructions can be fetched. The offset of the next instruction to be fetched from the code segment is contained in the instruction pointer register (EIP). After instruction fetch, the EIP register is incremented by the size of the instruction. A jump or call instruction can cause the EIP to change in three additional ways.

Instruction Pointer Relative The instruction contains an 8 or 32-bit displacement which is added to the EIP as a signed offset. The next instruction to be fetched is found at this new address. This is referred to as an intrasegment operation since it does not result in the segment registers being changed.

Direct Addressing The instruction contains a 16-bit value for the segment and a 32-bit value for the offset which are loaded into the CS and the EIP register respectively. This causes an absolute jump or call to any location in the 80386 memory. Since this operation results in a change to the segment register as well as the EIP, it is referred to as an intersegment operation.

Indirect Addressing The memory address referenced by the instruction is either a single 32-bit value or a 32-bit value and a 16-bit value. If a single 32-bit value exists, it is used to replace the value in the EIP. If a 16-bit value is also specified, it replaces the CS register.

11.6.2 Data Fetching

The following sections describe the addressing modes available on the 80386 processor.

11.6.2.1 Direct Addressing

Direct addressing is the simplest addressing mode on the 80386. The 16 or 32 bits following the operation code byte define an offset which when combined with the data segment register (DS) defines a memory address.

11.6.2.2 Immediate Addressing

Certain instructions can use data from the instruction itself. These operands are called immediate operands. Immediate operands may be 8, 16 or 32 bits.

11.6.2.3 Base Addressing

The offset of the operand is specified indirectly in one of the general registers. When ESP or EBP is specified as the base register, segment register SS is assumed, otherwise segment register DS is assumed.

11.6.2.4 Base + Displacement Addressing

This addressing mode consists of a register called the base and a displacement. The base register and displacement can be combined in two ways.

1. The displacement defines the offset of the beginning of an array. The base register contains the result of a computation which is the offset into the array of a particular element.
2. The displacement locates an item within a record and the base register points to a particular record.

11.6.2.5 (Index * Scale) + Displacement Addressing

This addressing mode consists of an index register, a scale factor and a displacement. The index register can be any of the general registers except ESP. This allows efficient indexing into a static array. The displacement defines the start of the array, the index register contains the subscript value, and the scale represents the size of an element in the array. The value of the scale factor can be 1, 2, 4 or 8.

11.6.2.6 Base + Index + Displacement Addressing

This addressing mode consists of a base register, an index register, and a displacement. This combination provides the following functions.

1. The two registers support a 2-dimensional array with the displacement defining the start of the array.
2. The two registers identify a particular record in an array of records with the displacement pointing to an item in the record.

11.6.2.7 Base + (Index * Scale) + Displacement Addressing

This addressing mode provides an efficient indexing of 2-dimensional arrays when the size of an array element is 1, 2, 4 or 8 bytes.

12 Pragmas

12.1 Introduction

A pragma is a compiler directive that provides the following capabilities.

1. Pragmas allow you to specify certain compiler options.
2. Pragmas can be used to direct the WATCOM C/386 code generator to emit specialized sequences of code for calling functions which use argument passing and value return techniques that differ from the default used by WATCOM C/386.
3. Pragmas can be used to describe attributes of functions (such as side effects) that are not possible at the C language level. The code generator can use this information to generate more efficient code.
4. Any sequence of in-line machine language instructions, including DOS and BIOS function calls, can be generated in the object code.

Pragmas are specified in the source file using the ***pragma*** directive. The following notation is used to describe the syntax of pragmas. In this chapter, this includes all definitions enclosed in a box.

ABC	The characters *abc* are required.
[abc]	The item *abc* is optional.
{abc}	The item *abc* may be repeated zero or more times.
'a'	The character *a* is required.
a|b|c	One of *a*, *b* or *c* may be specified.
a ::= b	The item *a* is defined in terms of *b*.
(a)	Item *a* is evaluated first.

A pragma has the following format.

```
#PRAGMA { option | library | pack | text | auxiliary } [';']
option ::= (ON | OFF) '(' {opt} ')'
        opt ::= UNREFERENCED | CHECK_STACK
library ::= LIBRARY '(' {library_name} ')'
        library_name ::= id | '"'{char}'"'
pack ::= PACK '(' [ ( 1 | 2 | 4) ] ')'
text ::= ALLOC_TEXT '(' seg_name',' id',' {','id} ')'
auxiliary ::= ( AUX | LINKAGE ) ( general_aux | alias_aux )
        general_aux ::= ['('alias_name')'] symbol_name [obj_name] {attributes}
        alias_aux ::= '(' symbol_name',' [FAR16] alias_name ')'
        alias_name ::= id
        symbol_name ::= DEFAULT | id
        obj_name ::= '"' {char|'*'|'^'} '"'
        attributes ::= ABORTS | FRAME | LOADDS | call_info | parm_info | return_info
                | modify_info
        call_info ::= ( ('='in_line) | FAR | NEAR )
        in_line ::= { constant | (SEG id) | (OFFSET id) | asm }
        parm_info ::= PARM { REVERSE | NOMEMORY | LOADDS | pop_info
                | {reg_set} }
        pop_info ::= CALLER | ROUTINE
        return_info ::= VALUE { NO8087 | struct_info | reg_set }
        struct_info ::= STRUCT { FLOAT | STRUCT | (ROUTINE | CALLER) | reg_set }
        modify_info ::= MODIFY { EXACT | NOMEMORY | reg_set }
        reg_set ::= '['{reg_name}']'
```

where:	*description:*
constant	is any valid C integer constant.
id	is any valid C identifier.
char	is any character.
reg_name	is a machine register or the string "8087".
seg_name	is a segment name.
asm	is an assembly language instruction or directive enclosed in double quotes.

The following classes of pragmas are supported.

1. pragmas that specify options
2. pragmas that specify default libraries
3. pragmas that describe the way structures are stored in memory

4. pragmas that provide auxiliary information used for code generation

12.2 Using Pragmas to Specify Options

Currently, the following options can be specified with pragmas:

1. unreferenced
2. check_stack

"unreferenced" controls the way WATCOM C/386 handles unused symbols. For example,

```
#pragma on (unreferenced);
```

will cause WATCOM C/386 to issue warning messages for all unused symbols. This is the default. Specifying

```
#pragma off (unreferenced);
```

will cause WATCOM C/386 to ignore unused symbols. Note that if the warning level is not high enough, warning messages for unused symbols will not be issued even if "unreferenced" was specified.

"check_stack" controls the way stack overflows are to be handled. For example,

```
#pragma on (check_stack);
```

will cause stack overflows to be detected and

```
#pragma off (check_stack);
```

will cause stack overflows to be ignored. When "check_stack" is on, WATCOM C/386 will generate a run-time call to a stack-checking routine at the start of every routine compiled. This run-time routine will issue an error if a stack overflow occurs when invoking the routine. The default is to check for stack overflows. Stack overflow checking is particularly useful when functions are invoked recursively. Note that if the stack overflows and stack checking has been suppressed, unpredictable results can occur.

If a stack overflow does occur during execution and you are sure that your program is not in error (i.e. it is not unnecessarily recursing), you must increase the stack size. This is done by linking your application again and specifying the "STACK" option to the WATCOM Linker with a larger stack size.

It is also possible to specify more than one option in a pragma as illustrated by the following example.

```
#pragma on (check_stack unreferenced);
```

12.3 Using Pragmas to Specify Default Libraries

Default libraries are specified in special object module records. Library names are extracted from these special records by the WATCOM Linker. When unresolved references remain after processing all object modules specified in linker "FILE" directives, these default libraries are searched after all libraries specified in linker "LIBRARY" directives have been searched.

By default, that is if no library pragma is specified, the WATCOM C/386 compiler generates, in the object file defining `main`, default libraries corresponding to the memory model and floating-point model used to compile the file. For example, if you have compiled the source file containing the main program for the flat memory model and the floating-point calls floating-point model, the libraries "clib3r" and "math3r" will be placed in the object file.

If you wish to add your own default libraries to this list, you can do so with a library pragma. Consider the following example.

```
#pragma library (mylib);
```

The name "mylib" will be added to the list of default libraries specified in the object file.

If the library specification contains characters such as '\', ':' or ',' (i.e., any character not allowed in a C identifier), you must enclose it in double quotes as in the following example.

```
#pragma library ("\penpoint\sdk\lib\graph.lib");
```

If you wish to specify more than one library in a library pragma you must separate them with spaces as in the following example.

```
#pragma library (mylib "\penpoint\sdk\lib\graph.lib");
```

12.4 Pack Pragmas

The pack pragma can be used to control the way in which structures are stored in memory. By default, WATCOM C/386 aligns all structures and its fields on a byte boundary. The following form of the pack pragma can be used to change the alignment of structures and their fields in memory.

#PRAGMA PACK '(' n ')' [';']

where: ***description:***

n is 1, 2, 4 or 8 and specifies the method of alignment.

The alignment of structure members is described in the following table. If the size of the member is 1, 2, 4 or 8, the alignment is given for each of the "zp" options. If the member of the structure is an array or structure, the alignment is described by the row "x".

sizeof(member)	zp1	zp2	zp4	zp8
1	0	0	0	0
2	0	2	2	2
4	0	2	4	4
8	0	2	4	8
x	aligned to largest member			

An alignment of 0 means no alignment, 2 means word boundary, 4 means doubleword boundary, etc. If the largest member of structure "x" is 1 byte then "x" is not aligned. If the largest member of structure "x" is 2 bytes then "x" is aligned according to row 2. If the largest member of structure "x" is 4 bytes then "x" is aligned according to row 4. If the largest member of structure "x" is 8 bytes then "x" is aligned according to row 8.

If no value is specified in the pack pragma, a default value of 1 is used. Note that the default value can be changed with the "zp" WATCOM C/386 compiler command line option.

12.5 The ALLOC_TEXT Pragma

The "alloc_text" pragma can be used to specify the name of the text segment into which the generated code for a function, or a list of functions, is to be placed. The following describes the form of the "alloc_text" pragma.

#PRAGMA ALLOC_TEXT '(' seg_name ',' fn { ',' fn } ')' [';']

where:	***description:***
seg_name	is the name of the text segment.
fn	is the name of a function.

Consider the following example.

```
#pragma alloc_text ( my_text, fn1, fn2 );
```

The code for the functions `fn1` and `fn2` will be placed in the segment `my_text`.

12.6 Auxiliary Pragmas

The following sections describe the capabilities provided by auxiliary pragmas.

12.6.1 Specifying a Symbol's Attributes

Auxiliary pragmas are used to describe attributes that affect code generation. Initially, the compiler defines a default set of attributes. Each auxiliary pragma refers to one of the following.

1. a symbol (such as a variable or function)
2. a type definition that resolves to a function type
3. the default set of attributes defined by the compiler.

When an auxiliary pragma refers to a particular symbol, a copy of the current set of default attributes is made and merged with the attributes specified in the auxiliary pragma. The resulting attributes are assigned to the specified symbol and can only be changed by another auxiliary pragma that refers to the same symbol.

An example of a type definition that resolves to a function type is the following.

```
typedef void (*func_type)();
```

When an auxiliary pragma refers to a such a type definition, a copy of the current set of default attributes is made and merged with the attributes specified in the auxiliary pragma. The resulting attributes are assigned to each function whose type matches the specified type definition.

When "default" is specified instead of a symbol name, the attributes specified by the auxiliary pragma change the default set of attributes. The resulting attributes are used by all symbols that have not been specifically referenced by a previous auxiliary pragma.

Note that all auxiliary pragmas are processed before code generation begins. Consider the following example.

```
code in which symbol x is referenced
#pragma aux y <attrs_1>;
code in which symbol y is referenced
code in which symbol z is referenced
#pragma aux default <attrs_2>;
#pragma aux x <attrs_3>;
```

Auxiliary attributes are assigned to symbols in the following way.

1. Symbol `x` is assigned the initial default attributes merged with the attributes specified by `<attrs_2>` and `<attrs_3>`.

2. Symbol `y` is assigned the initial default attributes merged with the attributes specified by `<attrs_1>`.

3. Symbol `z` is assigned the initial default attributes merged with the attributes specified by `<attrs_2>`.

12.6.2 Alias Names

When a symbol referred to by an auxiliary pragma includes an alias name, the attributes of the alias name are also assumed by the specified symbol.

There are two methods of specifying alias information. In the first method, the symbol assumes only the attributes of the alias name; no additional attributes can be specified. The second method is more general since it is possible to specify an alias name as well as

additional auxiliary information. In this case, the symbol assumes the attributes of the alias name as well as the attributes specified by the additional auxiliary information.

The simple form of the auxiliary pragma used to specify an alias is as follows.

#PRAGMA AUX '(' sym, [FAR16] alias ')' [';']

where:	***description:***
alias	is the alias name and is any valid C identifier.
sym	is any valid C identifier.

The `far16` attribute should only be used on systems that permit the calling of 16-bit code from 32-bit code. Currently, the only supported operating system that allows this is OS/2 2.x. If you have any libraries of functions or APIs that are only available as 16-bit code and you wish to access these functions and APIs from 32-bit code, you must specify the `far16` attribute. If the `far16` attribute is specified, the compiler will generate special code which allows the 16-bit code to be called from 32-bit code. Note that a `far16` function must be a function whose attributes are those specified by one of the alias names `cdecl` or `pascal`. These alias names will be described in a later section.

Consider the following example.

```
#PRAGMA aux push_args parm [] ;
#PRAGMA aux ( rtn, push_args ) ;
```

The routine `rtn` assumes the attributes of the alias name `push_args` which specifies that the arguments to `rtn` are passed on the stack.

Let us look at an example in which the symbol is a type definition.

```
typedef void (func_type)(int);

#pragma aux push_args parm [];
#pragma aux ( func_type, push_args );

extern func_type rtn1;
extern func_type rtn2;
```

The first auxiliary pragma defines an alias name called `push_args` that specifies the mechanism to be used to pass arguments. The mechanism is to pass all arguments on the stack. The second auxiliary pragma associates the attributes specified in the first pragma with

the type definition `func_type`. Since `rtn1` and `rtn2` are of type `func_type`, arguments to either of those functions will be passed on the stack.

The general form of an auxiliary pragma that can be used to specify an alias is as follows.

#PRAGMA AUX '(' alias ')' sym [';']

where:	***description:***
alias	is the alias name and is any valid C identifier.
sym	is any valid C identifier.

Consider the following example.

```
#pragma aux HIGH_C "*"                                  \
                   parm caller []                       \
                   value no8087                         \
                   modify [eax ecx edx fs gs];
#pragma aux (HIGH_C) rtn1;
#pragma aux (HIGH_C) rtn2;
#pragma aux (HIGH_C) rtn3;
```

The routines `rtn1`, `rtn2` and `rtn3` assume the same attributes as the alias name `HIGH_C` which defines the calling convention used by the MetaWare High C compiler. Note that register ES must also be specified in the "modify" register set when using a memory model that is not a small data model. Whenever calls are made to `rtn1`, `rtn2` and `rtn3`, the MetaWare High C calling convention will be used.

Note that if the attributes of `HIGH_C` change, only one pragma needs to be changed. If we had not used an alias name and specified the attributes in each of the three pragmas for `rtn1`, `rtn2` and `rtn3`, we would have to change all three pragmas. This approach also reduces the amount of memory required by the compiler to process the source file.

> ***WARNING!*** The alias name `HIGH_C` is just another symbol. If `HIGH_C` appeared in your source, it would assume the attributes specified in the pragma for `HIGH_C`

12.6.3 Special Symbols

A number of symbols are predefined by the compiler with a set of attributes that describe a particular calling convention. These symbols can be used as aliases. The following is a list of these symbols.

1. `cdecl` defines the calling convention used by Microsoft C.
2. `pascal` defines the calling convention used by OS/2 1.x and Windows API functions.
3. `syscall` defines the calling convention used by OS/2 2.x API functions.
4. `system` is identical to `syscall`.

The following describes the attributes of the above alias names.

```
#pragma aux pascal "^"                          \
            parm reverse routine []             \
            value struct float struct caller [] \
            modify [eax ebx ecx edx]
```

Notes:

1. All symbols are mapped to upper case.

2. Arguments are pushed on the stack in reverse order. That is, the first argument is pushed first, the second argument is pushed next, and so on. The routine being called will remove the arguments from the stack.

3. Floating-point values are returned in the same way as structures. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack immediately before the the call instruction. Upon returning from the call, register EAX will contain address of the space allocated for the return value.

4. Registers EAX, EBX, ECX and EDX are not saved and restored when a call is made.

```
#pragma aux cdecl "_*"                                  \
            parm caller []                              \
            value struct float struct routine [eax] \
            modify [eax ebx ecx edx]
```

Notes:

1. All symbols are preceded by an underscore character.

2. Arguments are pushed on the stack from left to right. That is, the last argument is pushed first. The calling routine will remove the arguments from the stack.

3. Floating-point values are returned in the same way as structures. When a structure is returned, the called routine allocates space for the return value and returns a pointer to the return value in register EAX

4. Registers EAX, EBX, ECX and EDX are not saved and restored when a call is made.

```
#pragma aux syscall "*"                    \
            parm caller []                 \
            value struct struct caller [] \
            modify [eax ebx ecx edx]
```

Notes:

1. Symbols names are not modified.

2. Arguments are pushed on the stack from left to right. That is, the last argument is pushed first. The calling routine will remove the arguments from the stack.

3. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack immediately before the the call instruction. Upon returning from the call, register EAX will contain address of the space allocated for the return value. Floating-point values are returned in 80x87 register ST(0).

4. Registers EAX, EBX, ECX and EDX are not saved and restored when a call is made.

12.6.4 Alternate Names for Symbols

The following form of the auxiliary pragma can be used to describe the mapping of a symbol from its source form to its object form.

#PRAGMA AUX sym obj_name [';']

where:	***description:***
sym	is any valid C identifier.
obj_name	is any character string enclosed in double quotes.

When specifying `obj_name`, the asterisk character ('*') has a special meaning; it is a placeholder for `sym`.

In the following example, the name "myrtn" will be replaced by "myrtn_" in the object file.

```
#pragma aux myrtn "*_";
```

This is the default for all function names.

In the following example, the name "myvar" will be replaced by "_myvar" in the object file.

```
#pragma aux myvar "_*";
```

This is the default for all variable names.

The default mapping for all symbols can also be changed as illustrated by the following example.

```
#pragma aux default "_*_";
```

The above auxiliary pragma specifies that all names will be prefixed and suffixed by an underscore character ('_').

The '^' character also has a special meaning. Whenever it is encountered in `obj_name`, it is replaced by the upper case version of `sym`.

In the following example, the name "myrtn" will be replaced by "MYRTN" in the object file.

```
#pragma aux myrtn "^";
```

12.6.5 Describing Calling Information

The following form of the auxiliary pragma can be used to describe the way a function is to be called.

```
#PRAGMA AUX sym FAR [';']
   or
#PRAGMA AUX sym NEAR [';']
   or
#PRAGMA AUX sym '=' in_line [';']

in_line ::= { constant | (SEG id) | (OFFSET id) | "asm" }
```

where:	*description:*
sym	is a function name.
constant	is a valid C integer constant.
id	is any valid C identifier.
SEG	specifies the segment of the symbol `id`.
OFFSET	specifies the offset of the symbol `id`.
asm	is an assembly language instruction or directive.

In the following example, WATCOM C/386 will generate a far call to the function `myrtn`.

```
#pragma aux myrtn far;
```

Note that this overrides the calling sequence that would normally be generated for a particular memory model. In other words, a far call will be generated even if you are compiling for a memory model with a small code model.

In the following example, WATCOM C/386 will generate a near call to the function `myrtn`.

```
#pragma aux myrtn near;
```

Note that this overrides the calling sequence that would normally be generated for a particular memory model. In other words, a near call will be generated even if you are compiling for a memory model with a big code model.

In the following example, WATCOM C/386 will generate the sequence of bytes following the "=" character in the auxiliary pragma whenever a call to `mode4` is encountered. `mode4` is called an in-line function.

```
void mode4(void);
#pragma aux mode4 =                    \
    0xb4 0x00        /* mov AH,0 */ \
    0xb0 0x04        /* mov AL,4 */ \
    0xcd 0x10        /* int 10H  */ \
    modify [ AH AL ];
```

The sequence in the above example represents the following lines of assembly language instructions.

```
mov    AH,0        ; select function "set mode"
mov    AL,4        ; specify mode (mode 4)
int    10H         ; BIOS video call
```

The above example demonstrates how to generate BIOS function calls in-line without writing an assembly language function and calling it from your C program. The C prototype for the function `mode4` is not necessary but is included so that we can take advantage of the argument type checking provided by WATCOM C/386.

The following example is equivalent to the above example but mnemonics for the assembly language instructions are used instead of the binary encoding of the assembly language instructions.

```
void mode4(void);
#pragma aux mode4 =      \
    "mov AH,0",     \
    "mov AL,4",     \
    "int 10H"            \
    modify [ AH AL ];
```

12.6.5.1 Loading Data Segment Register

An application may have been compiled so that the segment register DS does not contain the segment address of the default data segment (group "DGROUP"). This is usually the case if you are using a large data memory model. Suppose you wish to call a function that assumes that the segment register DS contains the segment address of the default data segment. It would be very cumbersome if you were forced to compile your application so that the segment register DS contained the default data segment (a small data memory model).

The following form of the auxiliary pragma will cause the segment register DS to be loaded with the segment address of the default data segment before calling the specified function.

```
#PRAGMA AUX sym PARM LOADDS [';']
```

where:	*description:*
sym	is a function name.

Alternatively, the following form of the auxiliary pragma will cause the segment register DS to be loaded with the segment address of the default data segment as part of the prologue sequence for the specified function.

```
#PRAGMA AUX sym LOADDS [';']
```

where:	*description:*
sym	is a function name.

12.6.5.2 Forcing a Stack Frame

Normally, a function contains a stack frame if arguments are passed on the stack or an automatic variable is allocated on the stack. No stack frame will be generated if the above conditions are not satisfied. The following form of the auxiliary pragma will force a stack frame to be generated under any circumstance.

```
#PRAGMA AUX sym FRAME [';']
```

where:	*description:*
sym	is a function name.

12.6.6 Describing Argument Information

Using auxiliary pragmas, you can describe the calling convention that WATCOM C/386 is to use for calling functions. This is particularly useful when interfacing to functions that have been compiled by other compilers or functions written in other programming languages.

The general form of an auxiliary pragma that describes argument passing is the following.

```
#PRAGMA AUX sym PARM { pop_info | REVERSE | {reg_set} } [';']

pop_info ::= CALLER | ROUTINE
```

where:	*description:*
sym	is a function name.
reg_set	is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

12.6.6.1 Passing Arguments in Registers

The following form of the auxiliary pragma can be used to specify the registers that are to be used to pass arguments to a particular function.

```
#PRAGMA AUX sym PARM {reg_set} [';']
```

where:	*description:*
sym	is a function name.
reg_set	is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

Register sets establish a priority for register allocation during argument list processing. Register sets are processed from left to right. However, within a register set, registers are chosen in any order. Once all register sets have been processed, any remaining arguments are pushed on the stack.

Note that regardless of the register sets specified, only certain combinations of registers will be selected for arguments of a particular type.

1. 8-byte arguments (arguments of type **double**) can only be passed in one of the following register pairs: EDX:EAX, ECX:EBX, ECX:EAX, ECX:ESI, EDX:EBX, EDI:EAX, ECX:EDI, EDX:ESI, EDI:EBX, ESI:EAX, ECX:EDX, EDX:EDI, EDI:ESI, ESI:EBX or EBX:EAX. Note that this does not include 8-byte structures. For example, if the following register set was specified for a routine having an argument of type **double,**

   ```
   [EBP EBX]
   ```

   the argument would be pushed on the stack since a valid register combination for 8-byte arguments is not contained in the register set.

2. A far pointer can only be passed in one of the following register pairs: DX:EAX, CX:EBX, CX:EAX, CX:ESI, DX:EBX, DI:EAX, CX:EDI, DX:ESI, DI:EBX, SI:EAX, CX:EDX, DX:EDI, DI:ESI, SI:EBX, BX:EAX, FS:ECX, FS:EDX, FS:EDI, FS:ESI, FS:EBX, FS:EAX, GS:ECX, GS:EDX, GS:EDI, GS:ESI, GS:EBX, GS:EAX, DS:ECX, DS:EDX, DS:EDI, DS:ESI, DS:EBX, DS:EAX, ES:ECX, ES:EDX, ES:EDI, ES:ESI, ES:EBX or ES:EAX. For example, if a far pointer is passed to a function with the following register set,

   ```
   [ES EBP]
   ```

   the argument would be pushed on the stack since a valid register combination for a far pointer is not contained in the register set.

3. The only registers that will be assigned to 4-byte arguments (i.e. arguments of type **int**) are: EAX, EBX, ECX, EDX, ESI and EDI. For example, if the following register set was specified for a routine with one argument of type **int,**

   ```
   [EBP]
   ```

   the argument would be pushed on the stack since a valid register combination for 4-byte arguments is not contained in the register set.

4. Arguments whose size is 1 byte or 2 bytes (arguments of type **char** and **short int**) are promoted to 4 bytes and are then assigned registers as if they were 4-byte arguments.

5. Arguments that do not fall into one of the above categories cannot be passed in registers and are passed on the stack. Once an argument has been assigned a

position on the stack, all remaining arguments will be assigned a position on the stack even if all register sets have not yet been exhausted.

Notes:

1. The default register set is [EAX EBX ECX EDX].

2. Specifying registers AH and AL is equivalent to specifying register AX. Specifying registers DH and DL is equivalent to specifying register DX. Specifying registers CH and CL is equivalent to specifying register CX. Specifying registers BH and BL is equivalent to specifying register BX. Specifying register EAX implies that register AX has been specified. Specifying register EBX implies that register BX has been specified. Specifying register ECX implies that register CX has been specified. Specifying register EDX implies that register DX has been specified. Specifying register EDI implies that register DI has been specified. Specifying register ESI implies that register SI has been specified. Specifying register EBP implies that register BP has been specified. Specifying register ESP implies that register SP has been specified.

3. If you are compiling for a memory model with a small data model, any register combination containing register DS becomes illegal. In a small data model, segment register DS must remain unchanged as it points to the program's data segment. Note that the "zdf" compiler option can be used to specify that register DS does not contain that segment address of the program's data segment. In this case, register combinations containing register DS are legal.

4. If you are compiling for the flat memory model, any register combination containing DS or ES becomes illegal. In a flat memory model, code and data reside in the same segment. Segment registers DS and ES point to this segment and must remain unchanged.

Consider the following example.

```
#pragma aux myrtn parm [eax ebx ecx edx] [ebp esi];
```

Suppose `myrtn` is a routine with 3 arguments each of type **double.**

1. The first argument will be passed in the register pair EDX:EAX.
2. The second argument will be passed in the register pair ECX:EBX.
3. The third argument will be passed on the stack since EBP:ESI is not a valid register pair for arguments of type **double.**

It is possible for registers from the second register set to be used before registers from the first register set are used. Consider the following example.

```
#pragma aux myrtn parm [eax ebx ecx edx] [esi edi];
```

Suppose `myrtn` is a routine with 3 arguments, the first of type **int** and the second and third of type **double.**

1. The first argument will be passed in the register EAX.
2. The second argument will be passed in the register pair ECX:EBX.
3. The third argument will be passed in the register set EDI:ESI.

Note that registers are no longer selected from a register set after registers are selected from subsequent register sets, even if all registers from the original register set have not been exhausted.

An empty register set is permitted. All subsequent register sets appearing after an empty register set are ignored; all remaining arguments are pushed on the stack.

Notes:

1. If a single empty register set is specified, all arguments are passed on the stack.

2. If no register set is specified, the default register set [EAX EBX ECX EDX] is used.

12.6.6.2 Forcing Arguments into Specific Registers

It is possible to force arguments into specific registers. Suppose you have a function, say "mycopy", that copies data. The first argument is the source, the second argument is the destination, and the third argument is the length to copy. If we want the first argument to be passed in the register ESI, the second argument to be passed in register EDI and the third argument to be passed in register ECX, the following auxiliary pragma can be used.

```
void mycopy( char near *, char *, int );
#pragma aux mycopy parm [EDI] [ESI] [ECX];
```

Note that you must be aware of the size of the arguments to ensure that the arguments get passed in the appropriate registers.

12.6.6.3 Passing Arguments to In-Line Functions

For functions whose code is generated by WATCOM C/386 and whose argument list is described by an auxiliary pragma, WATCOM C/386 has some freedom in choosing how arguments are assigned to registers. Since the code for in-line functions is specified by the programmer, the description of the argument list must be very explicit. To achieve this, WATCOM C/386 assumes that each register set corresponds to an argument. Consider the following in-line function called `scrollactivepgup`.

```
void scrollactivepgup(char,char,char,char,char,char);
#pragma aux scrollactivepgup =          \
    0xb4 0x06          /* mov AH,6 */   \
    0xcd 0x10          /* int 10h  */   \
    parm [ch] [cl] [dh] [dl] [al] [bh] \
    modify [ah];
```

The sequence in the above example represents the following lines of assembly language instructions.

```
mov    AH,6    ; select function "scroll active page up"
int    10h     ; BIOS video call
```

The BIOS video call to scroll the active page up requires the following arguments.

1. The row and column of the upper left corner of the scroll window is passed in registers CH and CL respectively.

2. The row and column of the lower right corner of the scroll window is passed in registers DH and DL respectively.

3. The number of lines blanked at the bottom of the window is passed in register AL.

4. The attribute to be used on the blank lines is passed in register BH.

When passing arguments, WATCOM C/386 will convert the argument so that it fits in the register(s) specified in the register set for that argument. For example, in the above example, if the first argument to `scrollactivepgup` was called with an argument whose type was **int** , it would first be converted to **char** before assigning it to register CH. Similarly, if an in-line function required its argument in register EAX and the argument was of type **short int** , the argument would be converted to **long int** before assigning it to register EAX.

In general, WATCOM C/386 assigns the following types to register sets.

1. A register set consisting of a single 8-bit register (1 byte) is assigned a type of **unsigned char.**

2. A register set consisting of a single 16-bit register (2 bytes) is assigned a type of **unsigned short int.**

3. A register set consisting of a single 32-bit register (4 bytes) is assigned a type of **unsigned long int.**

4. A register set consisting of two 32-bit registers (8 bytes) is assigned a type of **double.**

12.6.6.4 Removing Arguments from the Stack

The following form of the auxiliary pragma specifies who removes from the stack arguments that were passed on the stack.

```
#PRAGMA AUX sym PARM (CALLER | ROUTINE) [';']
```

where:	*description:*
sym	is a function name.

"caller" specifies that the caller will pop the arguments from the stack; "routine" specifies that the called routine will pop the arguments from the stack. If "caller" or "routine" is omitted, "routine" is assumed unless the default has been changed in a previous auxiliary pragma, in which case the new default is assumed.

12.6.6.5 Passing Arguments in Reverse Order

The following form of the auxiliary pragma specifies that arguments are passed in the reverse order.

```
#PRAGMA AUX sym PARM REVERSE [';']
```

where:	*description:*
sym	is a function name.

Normally, arguments are processed from left to right. The leftmost arguments are passed in registers and the rightmost arguments are passed on the stack (if the registers used for argument passing have been exhausted). Arguments that are passed on the stack are pushed from right to left.

When arguments are reversed, the rightmost arguments are passed in registers and the leftmost arguments are passed on the stack (if the registers used for argument passing have been exhausted). Arguments that are passed on the stack are pushed from left to right.

Reversing arguments is most useful for functions that require arguments to be passed on the stack in an order opposite from the default. The following auxiliary pragma demonstrates such a function.

```
#pragma aux rtn parm reverse [];
```

12.6.7 Describing Function Return Information

Using auxiliary pragmas, you can describe the way functions are to return values. This is particularly useful when interfacing to functions that have been compiled by other compilers or functions written in other programming languages.

The general form of an auxiliary pragma that describes the way a function returns its value is the following.

```
#PRAGMA AUX sym VALUE {NO8087 | reg_set | struct_info} [';']

struct_info ::= STRUCT
        {FLOAT | STRUCT | (ROUTINE | CALLER) | reg_set}
```

where:	*description:*
sym	is a function name.
reg_set	is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

12.6.7.1 Returning Function Values in Registers

The following form of the auxiliary pragma can be used to specify the registers that are to be used to return a function's value.

#PRAGMA AUX sym VALUE reg_set [';']

where:	***description:***
sym	is a function name.
reg_set	is a register set.

Depending on the type of the return value, only certain registers are allowed in *reg_set*.

1. For 1-byte return values, only the following registers are allowed: AL, AH, DL, DH, BL, BH, CL or CH. If no register set is specified, register AL will be used.

2. For 2-byte return values, only the following registers are allowed: AX, DX, BX, CX, SI or DI. If no register set is specified, register AX will be used.

3. For 4-byte return values (including near pointers), only the following register are allowed: EAX, EDX, EBX, ECX, ESI or EDI. If no register set is specified, register EAX will be used.

4. For functions that return far pointers, the following register pairs are allowed: DX:EAX, CX:EBX, CX:EAX, CX:ESI, DX:EBX, DI:EAX, CX:EDI, DX:ESI, DI:EBX, SI:EAX, CX:EDX, DX:EDI, DI:ESI, SI:EBX, BX:EAX, FS:ECX, FS:EDX, FS:EDI, FS:ESI, FS:EBX, FS:EAX, GS:ECX, GS:EDX, GS:EDI, GS:ESI, GS:EBX, GS:EAX, DS:ECX, DS:EDX, DS:EDI, DS:ESI, DS:EBX, DS:EAX, ES:ECX, ES:EDX, ES:EDI, ES:ESI, ES:EBX or ES:EAX. If no register set is specified, the registers DX:EAX will be used.

5. For 8-byte return values (functions of type **double**), only the following register pairs are allowed: EDX:EAX, ECX:EBX, ECX:EAX, ECX:ESI, EDX:EBX, EDI:EAX, ECX:EDI, EDX:ESI, EDI:EBX, ESI:EAX, ECX:EDX, EDX:EDI, EDI:ESI, ESI:EBX or EBX:EAX. If no register set is specified, the registers EDX:EAX will be used.

Notes:

1. An empty register set is not allowed.

2. If you are compiling for a memory model which has a small data model, any of the above register combinations containing register DS becomes illegal. In a small data model, segment register DS must remain unchanged as it points to the program's data segment.

3. If you are compiling for the flat memory model, any register combination containing DS or ES becomes illegal. In a flat memory model, code and data reside in the same segment. Segment registers DS and ES point to this segment and must remain unchanged.

12.6.7.2 Returning Structures

Typically, structures are not returned in registers. Instead, the caller allocates space on the stack for the return value and sets register ESI to point to it. The called routine then places the return value at the location pointed to by register ESI.

The following form of the auxiliary pragma can be used to specify the register that is to be used to point to the return value.

#PRAGMA AUX sym VALUE STRUCT (CALLER\|ROUTINE) reg_set [';']

where:	***description:***
sym	is a function name.
reg_set	is a register set.

"caller" specifies that the caller will allocate memory for the return value. The address of the memory allocated for the return value is placed in the register specified in the register set by the caller before the function is called. If an empty register set is specified, the address of the memory allocated for the return value will be pushed on the stack immediately before the call and will be returned in register EAX by the called routine.

"routine" specifies that the called routine will allocate memory for the return value. Upon returning to the caller, the register specified in the register set will contain the address of the return value. An empty register set is not allowed.

Only the following registers are allowed in the register set: EAX, EDX, EBX, ECX, ESI or EDI. Note that in a big data model, the address in the return register is assumed to be in the segment specified by the value in the SS segment register.

If the size of the structure being returned is 1, 2 or 4 bytes, it will be returned in registers. The return register will be selected from the register set in the following way.

1. A 1-byte structure will be returned in one of the following registers: AL, AH, DL, DH, BL, BH, CL or CH. If no register set is specified, register AL will be used.

2. A 2-byte structure will be returned in one of the following registers: AX, DX, BX, CX, SI or DI. If no register set is specified, register AX will be used.

3. A 4-byte structure will be returned in one of the following registers: EAX, EDX, EBX, ECX, ESI or EDI. If no register set is specified, register EAX will be used.

The following form of the auxiliary pragma can be used to specify that structures whose size is 1, 2 or 4 bytes are not to be returned in registers. Instead, the caller will allocate space on the stack for the structure return value and point register ESI to it.

#PRAGMA AUX sym VALUE STRUCT STRUCT [';']

where:	*description:*
sym	is a function name.

12.6.7.3 Returning Floating-Point Data

The following form of the auxiliary pragma can be used to specify that function return values whose type is **float** or **double** are not to be returned in registers. Instead, the caller will allocate space on the stack for the return value and point register ESI to it.

#PRAGMA AUX sym VALUE STRUCT FLOAT [';']

where:	*description:*
sym	is a function name.

In other words, floating-point values are to be returned in the same way structures are returned.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **float** or **double** are not to be returned in 80x87 registers when compiling with the "fpi", "fpi87", or "fpi287" option. Instead, the value will be returned in 80x86 registers. Function return values whose type is **float** will be returned in register EAX. Function return values whose type is **double** will be returned in registers EDX:EAX.

```
#PRAGMA AUX sym VALUE NO8087 [';']
```

where:	*description:*
sym	is a function name.

12.6.8 A Function that Never Returns

The following form of the auxiliary pragma can be used to describe a function that does not return to the caller.

```
#PRAGMA AUX sym ABORTS [';']
```

where:	*description:*
sym	is a function name.

Consider the following example.

```
#pragma aux exitrtn aborts;
extern void exitrtn(void);

void rtn()
  {
    exitrtn();
  }
```

`exitrtn` is defined to be a function that does not return. For example, it may call `exit` to return to the system. In this case, WATCOM C/386 generates a "jmp" instruction instead of a "call" instruction to invoke `exitrtn`.

12.6.9 Describing How Functions Use Memory

The following form of the auxiliary pragma can be used to describe a function that does not modify any memory (i.e., global or static variables) that is used directly or indirectly by the caller.

#PRAGMA AUX sym MODIFY NOMEMORY [';']

where:	***description:***
sym	is a function name.

Consider the following example.

```
#pragma off (check_stack);

extern void myrtn(void);

int i = { 1033 };

extern Rtn() {
    while( i < 10000 ) {
        i += 383;
    }
    myrtn();
    i += 13143;
};
```

To compile the above program, "rtn.c", we issue the following command.

```
C>wcc386 rtn /oai /d1
```

For illustrative purposes, we omit loop optimizations from the list of code optimizations that we want the compiler to perform. The "d1" compiler option is specified so that the object file produced by WATCOM C/386 contains source line information.

We can generate a file containing a disassembly of `rtnobj` by issuing the following command.

```
C>wdisasm rtn /l /s /r
```

The "s" option is specified so that the listing file produced by the WATCOM Disassembler contains source lines taken from `rtn.c`. The listing file `rtn.lst` appears as follows.

```
Module: rtn.c
Group: 'DGROUP' CONST,_DATA

Segment: '_TEXT' BYTE USE32  00000036 bytes

#pragma off (check_stack);

extern void myrtn(void);

int i = { 1033 };
```

```
extern Rtn() {
 0000  52                    Rtn_          push    EDX
 0001  8b 15 00 00 00 00                   mov     EDX,_i

    while( i < 10000 ) {
 0007  81 fa 10 27 00 00 L1                cmp     EDX,00002710H
 000d  7d 08                               jge     L2

        i += 383;
    }
 000f  81 c2 7f 01 00 00                   add     EDX,0000017fH
 0015  eb f0                               jmp     L1

    myrtn();
 0017  89 15 00 00 00 00 L2                mov     _i,EDX
 001d  e8 00 00 00 00                      call    myrtn_
 0022  8b 15 00 00 00 00                   mov     EDX,_i

    i += 13143;
 0028  81 c2 57 33 00 00                   add     EDX,00003357H
 002e  89 15 00 00 00 00                   mov     _i,EDX

}
 0034  5a                                  pop     EDX
 0035  c3                                  ret

No disassembly errors

-------------------------------------------------------------

Segment: '_DATA' WORD USE32  00000004 bytes
 0000  09 04 00 00              _i                - ....

No disassembly errors

-------------------------------------------------------------
```

Let us add the following auxiliary pragma to the source file.

```
#pragma aux myrtn modify nomemory;
```

If we compile the source file with the above pragma and disassemble the object file using the WATCOM Disassembler, we get the following listing file.

```
Module: rtn.c
Group: 'DGROUP' CONST,_DATA

Segment: '_TEXT' BYTE USE32  00000030 bytes

#pragma off (check_stack);
#pragma aux myrtn modify nomemory;
```

```
extern void myrtn(void);

int i = { 1033 };

extern Rtn() {
 0000  52                    Rtn_          push    EDX
 0001  8b 15 00 00 00 00                   mov     EDX,_i

    while( i < 10000 ) {
 0007  81 fa 10 27 00 00 L1                cmp     EDX,00002710H
 000d  7d 08                               jge     L2

        i += 383;
    }
 000f  81 c2 7f 01 00 00                   add     EDX,0000017fH
 0015  eb f0                               jmp     L1

    myrtn();
 0017  89 15 00 00 00 00 L2                mov     _i,EDX
 001d  e8 00 00 00 00                      call    myrtn_

    i += 13143;
 0022  81 c2 57 33 00 00                   add     EDX,00003357H
 0028  89 15 00 00 00 00                   mov     _i,EDX

}
 002e  5a                                  pop     EDX
 002f  c3                                  ret

No disassembly errors

-------------------------------------------------------------

Segment: '_DATA' WORD USE32  00000004 bytes
 0000  09 04 00 00              _i              - ....

No disassembly errors

-------------------------------------------------------------
```

Notice that the value of `i` is in register EDX after completion of the "while" loop. After the call to `myrtn`, the value of `i` is not loaded from memory into a register to perform the final addition. The auxiliary pragma informs the compiler that `myrtn` does not modify any memory (i.e., global or static variables) that is used directly or indirectly by `Rtn` and hence register EDX contains the correct value of `i`.

The preceding auxiliary pragma deals with routines that modify memory. Let us consider the case where routines reference memory. The following form of the auxiliary pragma can be used to describe a function that does not reference any memory (i.e., global or static variables) that is used directly or indirectly by the caller.

#PRAGMA AUX sym PARM NOMEMORY MODIFY NOMEMORY [';']

where: ***description:***

sym is a function name.

Notes:

1. You must specify both "parm nomemory" and "modify nomemory".

Let us replace the auxiliary pragma in the above example with the following auxiliary pragma.

```
#pragma aux myrtn parm nomemory modify nomemory;
```

If you now compile our source file and disassemble the object file using WDISASM, the result is the following listing file.

```
Module: rtn.c
Group: 'DGROUP' CONST,_DATA

Segment: '_TEXT' BYTE USE32  0000002a bytes

#pragma off (check_stack);
#pragma aux myrtn parm nomemory modify nomemory;

extern void myrtn(void);

int i = { 1033 };

extern Rtn() {
 0000  52                  Rtn_          push    EDX
 0001  8b 15 00 00 00 00                 mov     EDX,_i

    while( i < 10000 ) {
 0007  81 fa 10 27 00 00 L1              cmp     EDX,00002710H
 000d  7d 08                             jge     L2

        i += 383;
    }
 000f  81 c2 7f 01 00 00                 add     EDX,0000017fH
 0015  eb f0                             jmp     L1
```

```
    myrtn();
 0017  e8 00 00 00 00     L2              call    myrtn_

    i += 13143;
 001c  81 c2 57 33 00 00                  add     EDX,00003357H
 0022  89 15 00 00 00 00                  mov     _i,EDX

}
 0028  5a                                 pop     EDX
 0029  c3                                 ret

No disassembly errors

------------------------------------------------------------

Segment: '_DATA' WORD USE32  00000004 bytes
 0000  09 04 00 00              _i              - ....

No disassembly errors

------------------------------------------------------------
```

Notice that after completion of the "while" loop we did not have to update `i` with the value in register EDX before calling `myrtn`. The auxiliary pragma informs the compiler that `myrtn` does not reference any memory (i.e., global or static variables) that is used directly or indirectly by `myrtn` so updating `i` was not necessary before calling `myrtn`.

12.6.10 Describing the Registers Modified by a Function

The following form of the auxiliary pragma can be used to describe the registers that a function will use without saving.

#PRAGMA AUX sym MODIFY [EXACT] reg_set [';']

where:	***description:***
sym	is a function name.
reg_set	is a register set.

Specifying a register set informs WATCOM C/386 that the registers belonging to the register set are modified by the function. That is, the value in a register before calling the function is different from its value after execution of the function.

Registers that are used to pass arguments are assumed to be modified and hence do not have to be saved and restored by the called function. If necessary, the caller will contain code to save and restore the contents of registers used to pass arguments. Note that saving and restoring the contents of these registers may not be necessary if the called function does not modify them. The following form of the auxiliary pragma can be used to describe exactly those registers that will be modified by the called function.

#PRAGMA AUX sym MODIFY EXACT reg_set [';']

where: ***description:***

sym is a function name.

reg_set is a register set.

The above form of the auxiliary pragma tells WATCOM C/386 not to assume that the registers used to pass arguments will be modified by the called function. Instead, only the registers specified in the register set will be modified. This will prevent generation of the code which unnecessarily saves and restores the contents of the registers used to pass arguments.

12.6.11 An Example

As mentioned in an earlier section, the following pragma defines the calling convention for functions compiled by MetaWare's High C compiler.

```
#pragma aux HIGH_C "*"                          \
                   parm caller []               \
                   value no8087                 \
                   modify [eax ecx edx fs gs];
```

Note that register ES must also be specified in the "modify" register set when using a memory model with a non-small data model. Let us discuss this pragma in detail.

"*" specifies that all function and variable names appear in object form as they do in source form.

parm caller [] specifies that all arguments are to be passed on the stack (an empty register set was specified) and the caller will remove the arguments from the stack.

value no8087 specifies that floating-point values are to be returned using 80x86 registers and not 80x87 floating-point registers.

modify [eax ecx edx fs gs] specifies that registers EAX, ECX, EDX, FS and GS are not preserved by the called routine.

Note that the default method of returning integer values is used; 1-byte characters are returned in register AL, 2-byte integers are returned in register AX, and 4-byte integers are returned in register EAX.

12.6.12 Auxiliary Pragmas and the 80x87

This section deals with those aspects of auxiliary pragmas that are specific to the 80x87. The discussion in this chapter assumes that one of the "fpi", "fpi87", or "fpi287" options is used to compile functions. The following areas are affected by the use of these options.

1. passing floating-point arguments to functions,
2. returning floating-point values from functions and
3. which 80x87 floating-point registers are allowed to be modified by the called routine.

12.6.12.1 Using the 80x87 to Pass Arguments

By default, floating-point arguments are passed on the 80x86 stack. The 80x86 registers are never used to pass floating-point arguments when a function is compiled with the "fpi", "fpi87", or "fpi287" option. However, they can be used to pass arguments whose type is not floating-point such as arguments of type "int".

The following form of the auxiliary pragma can be used to describe the registers that are to be used to pass arguments to functions.

```
#PRAGMA AUX sym PARM {reg_set} [';']
```

where:	***description:***
sym	is a function name.
reg_set	is a register set. The register set can contain 80x86 registers and/or the string "8087".

Notes:

1. If an empty register set is specified, all arguments, including floating-point arguments, will be passed on the 80x86 stack.

When the string "8087" appears in a register set, it simply means that floating-point arguments can be passed in 80x87 floating-point registers if the source file is compiled with the "fpi", "fpi87", or "fpi287" option. Before discussing argument passing in detail, some general notes on the use of the 80x87 floating-point registers are given.

The 80x87 contains 8 floating-point registers which essentially form a stack. The stack pointer is called ST and is a number between 0 and 7 identifying which 80x87 floating-point register is at the top of the stack. ST is initially 0. 80x87 instructions reference these registers by specifying a floating-point register number. This number is then added to the current value of ST. The sum (taken modulo 8) specifies the 80x87 floating-point register to be used. The notation ST(n), where "n" is between 0 and 7, is used to refer to the position of an 80x87 floating-point register relative to ST.

When a floating-point value is loaded onto the 80x87 floating-point register stack, ST is decremented (modulo 8), and the value is loaded into ST(0). When a floating-point value is stored and popped from the 80x87 floating-point register stack, ST is incremented (modulo 8) and ST(1) becomes ST(0). The following illustrates the use of the 80x87 floating-point registers as a stack, assuming that the value of ST is 4 (4 values have been loaded onto the 80x87 floating-point register stack).

```
              +----------------+
        0     | 4th from top   |  ST(4)
              +----------------+
        1     | 5th from top   |  ST(5)
              +----------------+
        2     | 6th from top   |  ST(6)
              +----------------+
        3     | 7th from top   |  ST(7)
              +----------------+
ST -> 4       | top of stack   |  ST(0)
              +----------------+
        5     | 1st from top   |  ST(1)
              +----------------+
        6     | 2nd from top   |  ST(2)
              +----------------+
        7     | 3rd from top   |  ST(3)
              +----------------+
```

WATCOM C/386 uses only four of the eight 80x87 registers as a stack. These four registers are used to pass arguments. The other four registers form what is called the 80x87 cache. The cache is used for local floating-point variables.

Before a program begins execution, the state of the 80x87 registers is as follows.

1. The four 80x87 floating-point registers that form the stack are uninitialized.
2. The four 80x87 floating-point registers that form the 80x87 cache are initialized with zero.

Hence, initially the 80x87 cache is comprised of ST(0), ST(1), ST(2) and ST(3). ST has the value 4 as in the above diagram. When a floating-point value is pushed on the stack (as is the case when passing floating-point arguments), it becomes ST(0) and the 80x87 cache is comprised of ST(1), ST(2), ST(3) and ST(4). When the 80x87 stack is full, ST(0), ST(1), ST(2) and ST(3) form the stack and ST(4), ST(5), ST(6) and ST(7) form the 80x87 cache.

The rules for passing arguments are as follows.

1. If the argument is not floating-point, use the procedure described earlier in this chapter.

2. If the argument is floating-point, and a previous argument has been assigned a position on the 80x86 stack (instead of the 80x87 stack), the floating-point argument is also assigned a position on the 80x86 stack. Otherwise proceed to the next step.

3. If the 80x87 stack is not full (remember only 4 of the 8 floating-point registers form the stack), the floating-point argument is assigned floating-point register ST(0) (the top element of the 80x87 stack). The previous top element (if there was one) is now in ST(1). Since arguments are pushed on the stack from right to left, the leftmost floating-point argument will be in ST(0). Otherwise the floating-point argument is assigned a position on the 80x86 stack.

Consider the following example.

```
#pragma aux myrtn parm [8087];

void main() {
    float    x;
    double   y;
    int      i;
    long int j;

    x = 7.7;
    i = 7;
    y = 77.77;
    j = 77;
    myrtn( x, i, y, j );
}
```

`myrtn` is an assembly language function that requires four arguments. The first argument of type **float** (4 bytes), the second argument is of type **int** (4 bytes), the third argument is of type **double** (8 bytes) and the fourth argument is of type **long int** (4 bytes). These arguments will be passed to `myrtn` in the following way.

1. Since "8087" was specified in the register set, the first argument, being of type **float,** will be passed in an 80x87 floating-point register, leaving three 80x87 floating-point registers available for passing arguments.

2. The second argument will be passed on the stack since no 80x86 registers were specified in the register set.

3. The third argument will also be passed on the stack. Remember the following rule: once an argument is assigned a position on the stack, all remaining arguments will be assigned a position on the stack. Note that the above rule holds even though three 80x87 floating-point registers are available for passing floating-point arguments.

4. The fourth argument will also be passed on the stack.

Let us change the auxiliary pragma in the above example as follows.

```
#pragma aux myrtn parm [eax 8087];
```

The arguments will now be passed to `myrtn` in the following way.

1. Since "8087" was specified in the register set, the first argument, being of type "float" will be passed in an 80x87 floating-point register, leaving three 80x87 floating-point registers available for passing arguments.

2. The second argument will be passed in register EAX, exhausting the set of available 80x86 registers for argument passing.

3. The third argument, being of type "double", will also be passed in an 80x87 floating-point register, leaving two 80x87 floating-point registers available for passing arguments.

4. The fourth argument will be passed on the stack since no 80x86 registers remain in the register set.

12.6.12.2 Using the 80x87 to Return Function Values

The following form of the auxiliary pragma can be used to describe the way a function return value is returned.

```
#PRAGMA AUX sym VALUE reg_set [';']
```

where:	*description:*
sym	is a function name.
reg_set	is a register set.

If the register set contains the string "8087", floating-point values will be returned in 80x87 floating-point register ST(0). If "8087" is not specified in the register set, floating-point values will be returned on the stack. The caller will allocate space on the stack for the return value and point register ESI to it. No 80x86 registers will be used to return floating-point values when a function is compiled with the "fpi", "fpi87", or "fpi287" option.

12.6.12.3 Preserving 80x87 Floating-Point Registers Across Calls

As described in the chapter "Assembly Language Considerations", four of the eight 80x87 floating-point registers are used for a function's local variables. These four floating-point registers are known as the 80x87 cache. When a function is called, the registers in the 80x87 cache must be preserved. The following form of the auxiliary pragma specifies that the floating-point registers in the 80x87 cache may be modified by the specified function.

#PRAGMA AUX sym MODIFY reg_set [';']

where:	*description:*
sym	is a function name.
reg_set	is a register set containing the string "8087".

This instructs WATCOM C/386 to save any local variables that are located in the 80x87 cache before calling the specified routine.

Appendices

A. Use of Environment Variables

In the WATCOM C/386 software development package, a number of environment variables are used. This appendix summarizes their use with a particular component of the package.

FORCE The **FORCE** environment variable identifies a file that is to be included as part of the source input stream. This variable is used by WATCOM C/386.

```
SET FORCE=[d:][path]filename[.ext]
```

The specified file is included as if a

```
#include "[d:][path]filename[.ext]"
```

directive were placed at the start of the source file.

Example:

```
C>set force=\penpoint\sdk\inc\common.cnv
C>wcc report
```

The **FORCE** environment variable can be overridden by use of the WATCOM C/386 "fi" option.

INCLUDE The **INCLUDE** environment variable describes the location of the WATCOM C/386 and PenPoint header files (files with the ".h" filename extension). This variable is used by WATCOM C/386.

```
SET include=[d:][path];[d:][path]...
```

The **INCLUDE** environment string is like the **PATH** string in that you can specify one or more directories separated by semicolons (";").

LIB The **LIB** environment variable is used to select the libraries that will be used when the application is linked. This variable is used by the WATCOM Linker (WLINK.EXE). The **LIB** environment string is like the **PATH** string in that you can specify one or more directories separated by semicolons (";").

If you have the 286 development system, 16-bit applications can be linked for DOS, Microsoft Windows, OS/2, and QNX depending on which libraries are selected. If you have the 386 development system, 32-bit applications can be linked for DOS Extender systems and Microsoft Windows.

PATH

The **PATH** environment variable is used by DOS "COMMAND.COM" or OS/2 "CMD.EXE" to locate programs.

```
PATH [d:][path];[d:][path]...
```

The DOS or OS/2 **PATH** environment variable should include the disk and directory of the WATCOM C/386 binary program files such as "WCC386.EXE" when using WATCOM C/386 and its related tools.

If your host system is DOS:

The default installation directories for WATCOM C DOS binaries are called `\PENPOINT\SDK\BIN` and `\PENPOINT\SDK\BINB`.

Example:

```
C>path c:\penpoint\sdk\bin;c:\penpoint\sdk\binb
```

If your host system is OS/2:

The default installation directories for WATCOM C OS/2 binaries are called `\PENPOINT\SDK\BINP` and `\PENPOINT\SDK\BINB`.

Example:

```
[C:\]path c:\penpoint\sdk\binp;c:\penpoint\sdk\binb
```

The **PATH** environment variable is also used by the following programs in the described manner.

1. "WCL386.EXE" to locate the WATCOM C/386 compiler and the WATCOM Linker.
2. "WVIDEO.EXE" to locate programs and debugger command files.

TMP

The **TMP** environment variable describes the location (disk and path) for temporary files created by the WATCOM C/386 compiler and WATCOM Linker.

```
SET TMP=[d:][path]
```

Normally, WATCOM C/386 will create temporary spill files in the current directory. However, by defining the **TMP** environment variable to be a certain disk and directory, you can tell WATCOM C/386 where to place its temporary files. The same is true of the WATCOM Linker temporary file.

Consider the following definition of the **TMP** environment variable.

Example:

```
C>set tmp=d:\penpoint\sdk\tmp
```

The WATCOM C/386 compiler and WATCOM Linker will create its temporary files in `d:\penpoint\sdk\tmp`.

WATCOM

In order for the WATCOM Linker to locate the WATCOM C/386 for PenPoint 1.0 library files, the **WATCOM** environment variable should be defined. The **WATCOM** environment variable is used to locate the libraries that will be used when the application is linked. The default directory for WATCOM C/386 for PenPoint 1.0 files is "\PENPOINT\SDK".

Example:

```
C>set watcom=c:\penpoint\sdk
```

WCC386

The **WCC386** environment variable can be used to specify commonly used WATCOM C/386 options.

```
SET WCC386=/option1 /option2 ...
```

These options are processed before options specified on the command line. The following example defines the default options to be "d1" (include line number debug information in the object file) and "ox" (compile for maximum number of code optimizations).

Example:

```
C>set wcc386=/d1 /ox
```

Once the **WCC386** environment variable has been defined, those options listed become the default each time the WCC386 or WCC386P command is used.

WCL386

The **WCL386** environment variable can be used to specify commonly used WCL386 options.

```
SET WCL386=/option1 /option2 ...
```

These options are processed before options specified on the command line. The following example defines the default options to be "3s" (compile code for stack-based argument passing convention), "d1" (include line number debug information in the object file), and "ox" (compile for maximum number of code optimizations).

Example:

```
C>set wcl386=/3s /d1 /ox
```

Once the **WCL386** environment variable has been defined, those options listed become the default each time the WCL386 command is used.

WCGMEMORY The **WCGMEMORY** environment variable may be used to request a report of the amount of memory used by the compiler's code generator for its work area.

Example:

```
SET WCGMEMORY=?
```

When the memory amount is "?" then the code generator will report how much memory was used to generate the code.

It may also be used to instruct the compiler's code generator to allocate a fixed amount of memory for a work area.

Example:

```
SET WCGMEMORY=128
```

When the memory amount is "nnn" then exactly "nnnK" bytes will be used. In the above example, 128K bytes is requested. If less than "nnnK" is available then the compiler will quit with a fatal error message. If more than "nnnK" is available then only "nnnK" will be used.

There are two reasons why this second feature may be quite useful. In general, the more memory available to the code generator, the more optimal code it will generate. Thus, for two personal computers with different amounts of memory, the code generator may produce different (although correct) object code. If you have a software quality assurance requirement that the same results (i.e., code) be produced on two different machines then you should use this feature. To generate identical code on two personal computers with different memory configurations, you must ensure the following two conditions.

1. Both personal computers must have EMS memory or both computers must *not* have EMS memory.

2. You must set the **WCGMEMORY** environment variable identically on both machines.

The second reason where this feature is useful is on virtual memory paging systems (e.g., OS/2) where an unlimited amount of memory can be used by the code generator. If a very large module is being compiled, it may take a very long time to compile it. The code generator will continue to allocate more and more memory and cause an excessive amount of paging. By restricting the amount of memory that the code generator can use, you can reduce the amount of time required to compile a routine.

B. WATCOM C/386 Diagnostic Messages

The following is a list of all warning and error messages produced by WATCOM C/386. Diagnostic messages are issued during compilation and execution.

The messages listed in the following sections contain references to `%s`, `%d` and `%u`. They represent strings that are substituted by WATCOM C/386 to make the error message more exact. `%d` and `%u` represent a string of digits; `%s` any string, usually a symbolic name.

Consider the following program, named `err.c`, which contains errors.

```
#include <stdio.h>

void main()
  {
    int i;
    float i;

    i = 383;
    x = 13143.0;
    printf( "Integer value is %d\n", i );
    printf( "Floating-point value is %f\n", x );
  }
```

If we compile the above program, the following messages will appear on the screen.

```
err.c(6): Error! E1034: Symbol 'i' already defined
err.c(9): Error! E1011: Symbol 'x' has not been declared
err.c: 12 lines, included 191, 0 warnings, 2 errors
```

The diagnostic messages consist of the following information:

1. the name of the file being compiled,
2. the line number of the line containing the error (in parentheses),

3. a message number, and
4. text explaining the nature of the error.

In the above example, the first error occurred on line 6 of the file `err.c`. Error number 1034 (with the appropriate substitutions) was diagnosed. The second error occurred on line 9 of the file `err.c`. Error number 1011 (with the appropriate substitutions) was diagnosed.

The following sections contain a complete list of the messages. Run-time messages (messages displayed during execution) do not have message numbers associated with them.

B.1 Warning Level 1 Messages

W100 *Parameter %d contains inconsistent levels of indirection*

The function is expecting something like `char **` and it is being passed a `char *` for instance.

W101 *Non-portable pointer conversion*

This message is issued whenever you convert a non-zero constant to a pointer.

W102 *Type mismatch*

This message is issued for a function return value or an assignment where both types are pointers, but they are pointers to different kinds of objects.

W103 *Parameter count does not agree with previous definition*

You have either not enough parameters or too many parameters in a call to a function. If the function is supposed to have a variable number of parameters, then you can ignore this warning, or you can change the function declaration and prototypes to use the ",..." to indicate that the function indeed takes a variable number of parameters.

W104 *Inconsistent levels of indirection*

This occurs in an assignment or return statement when one of the operands has more levels of indirection than the other operand. For example, a `char **` is being assigned to a `char *`.

Solution: Correct the levels of indirection or use a `void *`.

W105 *Assignment found in boolean expression*

An assignment of a constant has been detected in a boolean expression. For example: "if(var = 0)". It is most likely that you want to use "==" for testing for equality.

W106 *Constant out of range* truncated

This message is issued if a constant cannot be represented in 32 bits or if a constant is outside the range of valid values that can be assigned to a variable.

W107 *Missing return value for function '%s'*

A function has been declared with a function return type, but no ***return*** statement was found in the function. Either add a ***return*** statement or change the function return type to ***void.***

W108 *Duplicate typedef already defined*

A duplicate typedef is not allowed in ANSI C. This warning is issued when compiling with extensions enabled. You should delete the duplicate typedef definition.

W110 *'fortran' pragma not defined*

You have used the ***fortran*** keyword in your program, but have not defined a #pragma for ***fortran.***

W111 *Meaningless use of an expression*

The line contains an expression that does nothing useful. In the example "i = (1,5);", the expression "1," is meaningless. This message is also generated for a comparison that is useless. Comparing an unsigned expression to see if it is >=, <, or <= to 0 is useless. Check to see if the expression should be signed instead of unsigned.

W112 *Pointer truncated*

A far pointer is being passed to a function that is expecting a near pointer, or a far pointer is being assigned to a near pointer.

W113 *Pointer type mismatch*

You have two pointers that either point to different objects, or the pointers are of different size, or they have different modifiers.

W114 *Missing ';'*

You are missing the semicolon ";" on the field definition just before the right curly brace "}".

W115 *&array may not produce intended result*

The type of the expression "&array" is different from the type of the expression "array". Suppose we have the declaration `char buffer[80]` Then the expression `(&buffer + 3)` will be evaluated as `(buffer + 3 * sizeof(buffer))` which is `(buffer + 3 * 80)` and not `(buffer + 3 * 1)` which is what most people expect to happen. The address of operator "&" is not required for getting the address of an array.

W116 *Attempt to return address of auto variable*

This warning usually indicates a serious programming error. When a function exits, the storage allocated on the stack for auto variables is released. This storage will be overwritten by further function calls and/or hardware interrupt service routines. Therefore, the data pointed to by the return value may be destroyed before your program has a chance to reference it or make a copy of it.

W117 *'##' tokens did not generate a single token (rest discarded)*

When two tokens are pasted together using ##, they must form a string that can be parsed as a single token.

W118 *Label '%s' has been defined but not referenced*

You have defined a label that is not referenced in a ***goto*** statement. It is possible that you are missing the ***case*** keyword when using an enumerated type name as a case in a ***switch*** statement. If not, then the label can be deleted.

B.2 Warning Level 2 Messages

W200 *'%s' has been referenced but never assigned a value*

You have used the variable in an expression without previously assigning a value to that variable.

W201 *Unreachable code*

The statement will never be executed, because there is no path through the program that causes control to reach this statement.

W202 *Symbol '%s' has been defined, but not referenced*

There are no references to the declared variable. The declaration for the variable can be deleted. If the variable is a parameter to a function, all calls to the function must also have the value for that parameter deleted.

In some cases, there may be a valid reason for retaining the variable. You can prevent the message from being issued through use of *#pragma off(unreferenced).*

W203 *Symbol '%s' has not been declared*

The symbol has been used in a preprocessor expression. The compiler assumes the symbol has a value of 0 and continues. A `#define` may be required for the symbol, or you may have forgotten to include the file which contains a `#define` for the symbol.

B.3 Warning Level 3 Messages

W300 *Nested comment found in comment started on line %u*

While scanning a comment for its end, the compiler detected `/*` for the start of another comment. Nested comments are not allowed in ANSI C. You may be missing the `*/` for the previous comment.

W301 *No prototype found for '%s'*

A reference for a function appears in your program, but you do not have a prototype for that function defined.

W302 *Expression is only useful for its side effects*

You have an expression that would have generated the warning "Meaningless use of an expression", except that it also contains a side-effect, such as ++, —, or a function call.

B.4 Error Messages

E1000 *BREAK must appear in while, do, for or switch statement*

A ***break*** statement has been found in an illegal place in the program. You may be missing an opening brace { for a ***while, do, for*** or ***switch*** statement.

E1001 *CASE must appear in switch statement*

A ***case*** label has been found that is not inside a ***switch*** statement.

E1002 *CONTINUE must appear in while, do or for statement*

The ***continue*** statement must be inside a ***while, do*** or ***for*** statement. You may have too many } between the ***while, do*** or ***for*** statement and the ***continue*** statement.

E1003 *DEFAULT must appear in switch statement*

A ***default*** label has been found that is not inside a ***switch*** statement. You may have too many } between the start of the ***switch*** and the ***default*** label.

E1004 *Misplaced '}' or missing earlier '{'*

An extra } has been found which cannot be matched up with an earlier { .

E1005 *Misplaced #elif directive*

The `#elif` directive must be inside an `#if` preprocessing group and before the `#else` directive if present.

E1006 *Misplaced #else directive*

The `#else` directive must be inside an `#if` preprocessing group and follow all `#elif` directives if present.

E1007 *Misplaced #endif directive*

A `#endif` preprocessing directive has been found without a matching `#if` directive. You either have an extra `#endif` or you are missing an `#if` directive earlier in the file.

E1008 *Only 1 DEFAULT per switch allowed*

You cannot have more than one ***default*** label in a ***switch*** statement.

E1009 *Expecting '%s' but found '%s'*

A syntax error has been detected. The tokens displayed in the message should help you to determine the problem.

E1010 *Type mismatch*

For pointer subtraction, both pointers must point to the same type. For other operators, both expressions must be assignment compatible.

E1011 *Symbol '%s' has not been declared*

The compiler has found a symbol which has not been previously declared. The symbol may be spelled differently than the declaration, or you may need to `#include` a header file that contains the declaration.

E1012 *Expression is not a function*

The compiler has found an expression that looks like a function call, but it is not defined as a function.

E1013 *Constant variable cannot be modified*

An expression or statement has been found which modifies a variable which has been declared with the ***const*** keyword.

E1014 *Left operand must be an 'lvalue'*

The operand on the left side of an "=" sign must be a variable or memory location which can have a value assigned to it.

E1015 *'%s' is already defined as a variable*

You are trying to declare a function with the same name as a previously declared variable.

E1016 *Expecting identifier*

The token following "->" and "." operators must be the name of an identifier which appears in the struct or union identified by the operand preceding the "->" and "." operators.

E1017 *Label '%s' already defined*

All labels within a function must be unique.

E1018 *Label '%s' not defined in function*

A ***goto*** statement has referenced a label that is not defined in the function. Add the necessary label or check the spelling of the label(s) in the function.

E1019 *Tag '%s' already defined*

All ***struct, union*** and ***enum*** tag names must be unique.

E1020 *Dimension cannot be 0*

The dimension of an array must be non-zero.

E1021 *Dimensions of multi-dimension array must be specified*

All dimensions of a multiple dimension array must be specified. The only exception is the first dimension which can declared as "[]".

E1022 *Missing or misspelled data type near '%s'*

The compiler has found an identifier that is not a predefined type or the name of a "typedef". Check the identifier for a spelling mistake.

E1023 *Storage class of parameter must be register or unspecified*

The only storage class allowed for a parameter declaration is ***register.***

E1024 *Declared symbol '%s' is not in parameter list*

Make sure that all the identifiers in the parameter list match those provided in the declarations between the start of the function and the opening brace "{".

E1025 *Parameter '%s' already declared*

A declaration for the specified parameter has already been processed.

E1026 *Invalid declarator*

A syntax error has occurred while parsing a declaration.

E1027 *Invalid storage class for function*

If a storage class is given for a function, it must be ***static*** or ***extern.***

E1028 *Variable '%s' cannot be void*

You cannot declare a ***void*** variable.

E1029 *Expression must be 'pointer to ...'*

An attempt has been made to de-reference (*) a variable or expression which is not declared to be a pointer.

E1030 *Cannot take the address of an rvalue*

You can only take the address of a variable or memory location.

E1031 *Name '%s' not found in struct/union %s*

The specified identifier is not one of the fields declared in the ***struct*** or ***union.*** Check that the field name is spelled correctly, or that you are pointing to the correct ***struct*** or ***union.***

E1032 *Expression for '.' must be a 'structure' or 'union'*

The compiler has encountered the pattern "expression" "." "field_name" where the expression is not a ***struct*** or ***union*** type.

E1033 *Expression for '->' must be 'pointer to struct or union'*

The compiler has encountered the pattern "expression" "->" "field_name" where the expression is not a pointer to ***struct*** or ***union*** type.

E1034 *Symbol '%s' already defined*

The specified symbol has already been defined.

E1035 *static function '%s' has not been defined*

A prototype has been found for a ***static*** function, but a definition for the ***static*** function has not been found in the file.

E1036 *Right operand of '%s' is a pointer*

The right operand of "+=" and "–=" cannot be a pointer. The right operand of "–" cannot be a pointer unless the left operand is also a pointer.

E1037 *Type cast must be a scalar type*

You cannot type cast an expression to be a ***struct, union,*** array or function.

E1038 *Expecting label for goto statement*

The ***goto*** statement requires the name of a label.

E1039 *Duplicate case value '%d' found*

Every case value in a ***switch*** statement must be unique.

E1040 *Field width too large*

The maximum field width allowed is 16 bits.

E1041 *Field width of 0 with symbol not allowed*

A bit field must be at least one bit in size.

E1042 *Field width must be positive*

You cannot have a negative field width.

E1043 *Invalid type specified for bit field*

The types allowed for bit fields are ***signed*** or ***unsigned*** varieties of ***char, short*** and ***int.***

E1044 *Variable '%s' has incomplete type*

A full definition of a ***struct*** or ***union*** has not been given.

E1045 *Subscript on non-array*

One of the operands of "[]" must be an array.

E1046 *Incomplete comment*

The compiler did not find `*/` to mark the end of a comment.

E1047 *Argument for # must be a macro parm*

The argument for the stringize operator "#" must be a macro parameter.

E1048 *Unknown preprocessing directive '#%s'*

An unrecognized preprocessing directive has been encountered. Check for correct spelling.

E1049 *Invalid #include directive*

A syntax error has been encountered in a `#include` directive.

E1050 *Not enough parameters given for macro '%s'*

You have not supplied enough parameters to the specified macro.

E1051 *Not expecting a return value for function '%s'*

The specified function is declared as a ***void*** function. Delete the ***return*** statement, or change the type of the function.

E1052 *Expression has void type*

You tried to use the value of a ***void*** expression inside another expression.

E1053 *Cannot take the address of a bit field*

The smallest addressable unit is a byte. You cannot take the address of a bit field.

E1054 *Expression must be constant*

The compiler expects a constant expression. This message can occur during static initialization if you are trying to initialize a non-pointer type with an address expression.

E1055 *Unable to open '%s'*

The file specified in an `#include` directive could not be located. Make sure that the file name is spelled correctly, or that the appropriate path for the file is included in the list of paths specified in the `INCLUDE` environment variable or the "i=" option on the command line.

E1056 *Too many parameters given for macro '%s'*

You have supplied too many parameters for the specified macro.

E1057 *Modifiers disagree with previous definition of '%s'*

You have more than one definition or prototype for the variable or function which have different type modifiers.

E1058 *Cannot use typedef '%s' as a variable*

The name of a typedef has been found when an operand or operator is expected. If you are trying to use a type cast, make sure there are parentheses around the type, otherwise check for a spelling mistake.

E1059 *Invalid storage class for non-local variable*

A variable with module scope cannot be defined with the storage class of ***auto*** or ***register.***

E1060 *Invalid type*

An invalid combination of the following keywords has been specified in a type declaration: ***const, volatile, signed, unsigned, char, int, short, long, float*** and ***double.***

E1061 *Expecting data or function declaration, but found '%s'*

The compiler is expecting the start of a data or function declaration. If you are only part way through a function, then you have too many closing braces "}".

E1062 *Inconsistent return type for function '%s'*

Two prototypes for the same function disagree.

E1063 *Missing operand*

An operand is required in the expression being parsed.

E1064 *Out of memory*

The compiler has run out of memory to store information about the file being compiled. Try reducing the number of data declarations and or the size of the file being compiled. Do not `#include` header files that are not required.

E1065 *Invalid character constant*

This message is issued for an improperly formed character constant.

E1066 *Cannot perform operation with pointer to void*

You cannot use a "pointer to void" with the operators +, –, ++, —, += and –=.

E1067 *Cannot take address of variable with storage class 'register'*

If you want to take the address of a local variable, change the storage class from ***register*** to ***auto.***

E1068 *Variable '%s' already initialized*

The specified variable has already been statically initialized.

E1069 *Ending " missing for string literal*

The compiler did not find a second double quote to end the string literal.

E1070 *Data for aggregate type must be enclosed in curly braces*

When an array, struct or union is statically initialized, the data must be enclosed in curly braces { }.

E1071 *Type of parameter %d does not match previous definition*

The type of the specified parameter is incompatible with the prototype for that function. The following example illustrates a problem that can arise when the sequence of declarations is in the wrong order.

Example:

```
/* Uncommenting the following line will
   eliminate the error */
/* struct foo; */

void fn1( struct foo * );

struct foo {
    int     a,b;
};

void fn1( struct foo *bar )
{
    fn2( bar );
}
```

The problem can be corrected by reordering the sequence in which items are declared (by moving the description of the structure `foo` ahead of its first reference or by adding the indicated statement). This will assure that the first instance of structure `foo` is defined at the proper outer scope.

E1072 *Storage class disagrees with previous definition of '%s'*

The previous definition of the specified variable has a storage class of ***static.*** The current definition must have a storage class of ***static*** or ***extern.***

E1073 *Invalid option '%s'*

The specified option is not recognized by the compiler.

E1074 *Invalid optimization option '%s'*

The specified option is an unrecognized optimization option.

E1075 *Invalid memory model '%s'*

Memory model option must be one of "ms", "mm", "mc", "ml", "mh" or "mf" which selects the Small, Medium, Compact, Large, Huge or Flat memory model.

E1076 *Missing semicolon at end of declaration*

You are missing a semicolon ";" on the declaration just before the left curly brace "{".

E1077 *Missing '}'*

The compiler detected end of file before finding a right curly brace "}" to end the current function.

E1078 *Invalid type for switch expression*

The type of a switch expression must be integral.

E1079 *Expression must be integral*

An integral expression is required.

E1080 *Expression must be arithmetic*

Both operands of the "*", "/" and "%" operators must be arithmetic. The operand of the unary minus must also be arithmetic.

E1081 *Expression must be scalar type*

A scalar expression is required.

E1082 *Statement required after label*

The C language definition requires a statement following a label. You can use a null statement which consists of just a semicolon (";").

E1083 *Statement required after 'do'*

A statement is required between the ***do*** and ***while*** keywords.

E1084 *Statement required after 'case'*

The C language definition requires a statement following a ***case*** label. You can use a null statement which consists of just a semicolon (";").

E1085 *Statement required after 'default'*

The C language definition requires a statement following a ***default*** label. You can use a null statement which consists of just a semicolon (";").

E1086 *Expression too complicated, split it up and try again*

The expression contains too many levels of nested parentheses. Divide the expression up into two or more sub-expressions.

E1087 *Missing matching #endif directive*

You are missing a `#endif` to terminate a `#if`, `#ifdef` or `#ifndef` preprocessing directive.

E1088 *Invalid macro definition, missing)*

The right parenthesis ")" is required for a function-like macro definition.

E1089 *Missing) for expansion of '%s' macro*

The compiler encountered end-of-file while collecting up the argument for a function-like macro. A right parenthesis ")" is required to mark the end of the argument(s) for a function-like macro.

E1090 *Invalid conversion*

A ***struct*** or ***union*** cannot be converted to anything. A ***float*** or ***double*** cannot be converted to a pointer and a pointer cannot be converted to a ***float*** or ***double.***

E1091 *%s*

This is a user message generated with the `#error` preprocessing directive.

E1092 *Cannot define an array of functions*

You can have an array of pointers to functions, but not an array of functions.

E1093 *Function cannot return an array*

A function cannot return an array. You can return a pointer to an array.

E1094 *Function cannot return a function*

You cannot return a function. You can return a pointer to a function.

E1095 *Cannot take address of local variable in static initialization*

You cannot take the address of an ***auto*** variable at compile time.

E1096 *Inconsistent use of return statements*

The compiler has found a ***return*** statement which returns a value and a ***return*** statement that does not return a value both in the same function. The ***return***

statement which does not return a value needs to have a value specified to be consistent with the other ***return*** statement in the function.

E1097 *Missing ? or misplaced :*

The compiler has detected a syntax error related to the "?" and ":" operators. You may need parenthesis around the expressions involved so that it can be parsed correctly.

E1098 *Maximum struct or union size is 64K*

The size of a ***struct*** or ***union*** is limited to 64K so that the compiler can represent the offset of a member in a 16-bit register.

E1099 *Statement must be inside function* Probable cause: missing {

The compiler has detected a statement such as ***for, while, switch,*** etc., which must be inside a function. You either have too many closing braces "}" or you are missing an opening brace "{" earlier in the function.

E1100 *Definition of macro '%s' not identical to previous definition*

If a macro is defined more than once, the definitions must be identical. If you want to redefine a macro to have a different definition, you must `#undef` it before you can define it with a new definition.

E1101 *Cannot #undef __LINE__, __FILE__, __DATE__, __TIME__, __STDC__ or* 'defined'

The special macros `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, and `__STDC__`, and the identifier "defined", cannot be deleted by the `#undef` directive.

E1102 *Cannot #define the name 'defined'*

You cannot define a macro called `defined`.

E1103 *## must not be at start or end of replacement tokens*

There must be a token on each side of the "##" (token pasting) operator.

E1104 *Type cast not allowed in #if or #elif expression*

A type cast is not allowed in a preprocessor expression.

E1105 *'sizeof' not allowed in #if or #elif expression*

The ***sizeof*** operator is not allowed in a preprocessor expression.

E1106 *Cannot compare a struct or union*

A ***struct*** or ***union*** cannot be compared with "==" or "!=". You must compare each member of a ***struct*** or ***union*** to determine equality or inequality. If the ***struct*** or ***union*** is packed (has no holes in it for alignment purposes) then you can compare two structs using `memcmp`.

E1107 *Enumerator list cannot be empty*

You must have at least one identifier in an ***enum*** list.

E1108 *Invalid floating-point constant*

The exponent part of the floating-point constant is not formed correctly.

E1109 *Cannot take sizeof a bit field*

The smallest object that you can ask for the size of is a char.

E1110 *Cannot initialize variable with storage class of extern*

A storage class of ***extern*** is used to associate the variable with its actual definition somewhere else in the program.

E1111 *Invalid storage class for parameter*

The only storage class allowed for a parameter is ***register.***

E1112 *Initializer list cannot be empty*

An initializer list must have at least one item specified.

E1113 *Expression has incomplete type*

An attempt has been made to access a struct or union whose definition is not known, or an array whose dimensions are not known.

E1114 *Struct or union cannot contain itself*

You cannot have a ***struct*** or ***union*** contain itself. You can have a pointer in the ***struct*** which points to an instance of itself. Check for a missing "*" in the declaration.

E1115 *Incomplete enum declaration*

The enumeration tag has not been previously defined.

E1116 *An id list not allowed except for function definition*

A function prototype must contain type information.

E1117 *Must use 'va_start' macro inside function with variable parameters*

The `va_start` macro is used to setup access to the parameters in a function that takes a variable number of parameters. A function is defined with a variable number of parameters by declaring the last parameter in the function as "...".

E1118 ****FATAL*** %s*

A fatal error has been detected during code generation time. The type of error is displayed in the message.

E1119 *Internal compiler error %d*

A bug has been encountered in the WATCOM C compiler. Please report the specified internal compiler error number and any other helpful details about the program being compiled to WATCOM so that we can fix the problem.

E1120 *Parameter number %d - invalid register in #pragma*

The designated registers cannot hold the value for the parameter.

E1121 *Procedure '%s' has invalid return register in #pragma*

The size of the return register does not match the size of the result returned by the function.

E1122 *Illegal register modified by '%s' #pragma*

The EBP, CS, DS, ES, and SS registers cannot be modified in flat memory models. The EBP, CS, DS, and SS registers cannot be modified in small data

models. The EBP, CS, and SS registers cannot be modified in large data models.

E1123 *File must contain at least one external definition*

Every file must contain at least one global object, (either a data variable or a function). This message is only issued in strict ANSI mode (-za).

E1124 *Out of macro space*

The compiler ran out of memory for storing macro definitions.

E1125 *Keyboard interrupt detected*

The compile has been aborted with Ctrl/C or Ctrl/Break.

E1126 *Array, struct or union cannot be placed in a register*

Only scalar objects can be specified with the ***register*** class.

E1127 *Type required in parameter list*

If the first parameter in a function definition or prototype is defined with a type, then all of the parameters must have a type specified.

E1128 *Enum constant too large*

All of the constants must fit in either an ***int*** or ***unsigned.***

E1129 *Type does not agree with previous definition of '%s'*

You have more than one definition of a variable or function that do not agree.

E1130 *Duplicate name '%s' not allowed in struct or union*

All the field names in a ***struct*** or ***union*** must be unique.

E1131 *Duplicate macro parameter '%s'*

The parameters specified in a macro definition must be unique.

E1132 *Unable to open work file*

The compiler tries to open a new work file by the name "__wrkN__.tmp" where N is the digit 0 to 9. This message will be issued if all of those files already exist.

E1133 *Write error on work file*

An error was encountered trying to write information to the work file. The disk could be full.

E1134 *Read error on work file*

An error was encountered trying to read information from the work file.

E1135 *Seek error on work file*

An error was encountered trying to seek to a position in the work file.

E1136 *Token too long* truncated

The token must be less than 510 bytes in length.

E1137 *Out of enum space*

The compiler has run out of space allocated to store information on all of the ***enum*** constants defined in your program.

E1138 *Filename required on command line*

The name of a file to be compiled must be specified on the command line.

E1139 *Command line contains more than one file to compile*

You have more than one file name specified on the command line to be compiled. The compiler can only compile one file at a time. You can use the "Compile and Link" driver program ***WCL386*** to compile multiple files with a single command.

E1140 *Unrecognized database engine '%s'*

This message only applies to the version of the compiler containing embedded SQL support. Correct the value specified on the command line.

E1141 *Unrecognized SQL language level '%s'*

This message only applies to the version of the compiler containing embedded SQL support. Correct the value specified on the command line.

E1142 *Too many bytes specified in #pragma*

There is an internal limit on the number of bytes for inline code that can be specified with a pragma. Try splitting the function into two or more smaller functions.

E1143 *Cannot resolve linkage conventions for routine '%s' #pragma*

The compiler cannot generate correct code for the specified routine because of register conflicts. Change the registers used by the parameters of the pragma.

E1144 *Symbol '%s' in pragma must be global*

The inline code for a pragma can only reference a global variable or function. You can only reference a parameter or local variable by passing it as a parameter to the inline code pragma.

E1145 *Internal compiler limit exceeded, break module into smaller pieces*

The compiler can handle 65535 quadruples, 65535 leaves, and 65535 symbol table entries and literal strings. If you exceed one of these limits, the program must be broken into smaller pieces until it is capable of being processed by the compiler.

E1146 *Invalid initializer for integer data type*

Integer data types (int and long) can be initialized with numeric expressions or address expressions that are the same size as the integer data type being initialized.

E1147 *Too many errors: compilation aborted*

The compiler stops compiling when the number of errors generated exceeds the error limit. The error limit can be set with the "-e" option. The default error limit is 20.

E1148 *Not enough memory for code generator*

The DOS real mode compiler was unable to start the code generator phase of the compilation because there was not enough memory. Free up as much memory as possible by loading TSR's into high memory, etc. until there is enough

memory to run the code generator, or use the protected mode compiler on a 386 or 486 machine.

E1149 *Unable to execute '%s'*

The DOS real mode compiler could not start the code generator because it could not find it, or some other error occurred when trying to start it. Make sure the code generator is in the PATH. The name of the real mode code generator is I86WCGL.EXE for the 16-bit compiler, and 386WCGL.EXE for the 32-bit compiler. It is possible that there wasn't enough memory to start the code generator (see previous error message description).

E1150 *Incompatible code generator*

The code generator is incompatible with the C front end. This means you are either using a newer C front end with an older version of the code generator, or an older C front end with a newer version of the code generator.

E1151 *Parameter count does not agree with previous definition*

You have either not enough parameters or too many parameters in a call to a function. If the function is supposed to have a variable number of parameters, then you are missing the ", ..." in the function prototype.

B.5 Informational Messages

I2000 *Not enough memory to fully optimize procedure '%s'*

The compiler did not have enough memory to fully optimize the specified procedure.

On non-386 systems, do anything you can to make as much memory available before starting the compile. For example, unload all unnecessary terminate and stay resident utilities.

If you are using a 386 machine, you can use any of the utility programs available that makes some of the memory above 640K available to DOS programs. If you have extended memory, switch to the protected-mode version of the compiler.

I2001 *Not enough memory to maintain full peephole*

This message is similar to message I2000. It is only printed if the warning level is greater than or equal to 4. The main reason for this message is for those people who are concerned about reproducing the exact same object code when the same source file is compiled on a different machine. You may not be able to reproduce the exact same object code from one compile to the next unless the available memory is exactly the same.

Run-Time Error Messages

1. Assertion failed: %s, file %s, line %d

   This message is displayed whenever an assertion that you have made in your program is not true.

2. Stack Overflow!

   Your program is trying to use more stack space than is available. If you believe that your program is correct, you can increase the size of the stack by using the "option stack=nnnn" when you link the program. The stack size can also be specified with the "k" option if you are using WCL386.

3. Floating-point support not loaded

   You have called one of the printf functions with a format of "%e", "%f", or "%g", but have not passed a floating-point value. The compiler generates a reference to the variable "_fltused_" whenever you pass a floating-point value to a function. During the linking phase, the extra floating-point formatting routines will also be brought into your application when "_fltused_" is referenced. Otherwise, you only get the non floating-point formatting routines.

4. *** NULL assignment detected

   This message is displayed if any of the first 32 bytes of your program's data segment has been modified. The check is performed just before your program exits to the operating system. All this message means is that sometime during the execution of your program, this memory was modified.

   To find the problem, you must link your application with debugging information and use VIDEO to monitor its execution. First, run the application with VIDEO until it completes. Examine the first 16 bytes of the data segment ("examine __nullarea") and press the space bar to see the next 16 bytes. Any values that are not equal to '01' have been modified. Reload the application, set watch points on the modified locations, and start execution. VIDEO will stop when the specified location(s) change in value.

B.6 errno Values and Their Meanings

The following errors can be generated by the C run-time library. These error codes correspond to the error types defined in `errno.h`.

ENOENT *No such file or directory*

The specified file or directory cannot be found.

E2BIG *Argument list too big*

The argument list passed to the `spawn...`, `exec...` or `system` functions requires more than 128 bytes, or the environment information exceeds 32K.

ENOEXEC *Exec format error*

The executable file has an invalid format.

EBADF *Bad file number*

The file handle is not a valid file handle value or it does not correspond to an open file.

ENOMEM *Not enough memory*

There was not enough memory available to perform the specified request.

EACCES *Permission denied*

You do not have the required (or correct) permissions to access a file.

EEXIST *File exists*

An attempt was made to create a file with the O_EXCL (exclusive) flag when the file already exists.

EXDEV *Cross-device link*

An attempt was made to rename a file to a different device.

EINVAL *Invalid argument*

An invalid value was specified for one of the arguments to a function.

ENFILE *File table overflow*

All the FILE structures are in use, so no more files can be opened.

EMFILE *Too many open files*

There are no more file handles available, so no more files can be opened. The maximum number of file handles available is controlled by the "FILES=" option in the "CONFIG.SYS" file.

ENOSPC *No space left on device*

No more space is left for writing on the device, which usually means that the disk is full.

EDOM *Argument too large*

An argument to a math function is not in the domain of the function.

ERANGE *Result too large*

The result of a math function could not be represented (too small, or too large).

EDEADLK *Resource deadlock would occur*

A resource deadlock would occur with regards to locked files.

B.7 Math Run-Time Error Messages

The following errors can be generated by the math functions in the C run-time library. These error codes correspond to the exception types defined in `math.h` and returned by the `matherr` function when a math error occurs.

DOMAIN *Domain error*

An argument to the function is outside the domain of the function.

OVERFLOW *Overflow range error*

The function result is too large.

PLOSS *Partial loss of significance*

A partial loss of significance occurred.

SING *Argument singularity*

An argument to the function has a bad value (e.g., log(0.0)).

TLOSS *Total loss of significance*

A total loss of significance occurred. An argument to a function was too large to produce a meaningful result.

UNDERFLOW *Underflow range error*

The result is too small to be represented.

C. Commonly Asked Questions and Answers

C.1 Introduction

As with any sophisticated piece of software, there are topics that are not directly addressed by the descriptive portions of the manuals. The purpose of this appendix is to anticipate common questions concerning WATCOM C/386. It is difficult to predict what topics will prove to be useful but with that in mind, we hope that this appendix will help our customers make full use of WATCOM C/386.

The following topics are discussed in this appendix:

- Why can't the compiler find "stdio.h"?
- What do these undefined references mean?
- Why aren't my variables set to zero?
- How can I see my source files in the debugger?
- What does "Stack Overflow!" mean?
- Why do I get redefinition errors from WLINK?
- How do I convert to WATCOM C/386?
- What should I know about optimization?

C.2 Why can't the compiler find "stdio.h"?

The standard header files are usually located in the sub-directory that the WATCOM C/386 compiler is installed in. Suppose that the header files are located in the sub-directory `c:\penpoint\sdk\inc`. If the compiler indicates (through an error message) that it is unable to locate the file `stdio.h`, we have forgotten something. There are two ways to indicate to the WATCOM C/386 compiler the location of the standard header files.

1. use the **INCLUDE** environment variable
2. use the "i" option (WATCOM C/386, WCL)

The use of the environment variable is the simplest way to ensure that the include files will be found. For instance, if you include the following line in your system initialization file, `autoexec.bat`

```
set include=c:\penpoint\sdk\inc
```

the WATCOM C/386 compiler will be able to find the standard include files. The use of the "i" option is another way to give the directory name of the standard include files.

Example:

```
C>wcc386 myfile.c -ic:\penpoint\sdk\inc
```

The usual manner that these methods are combined is as follows. The **INCLUDE** environment variable is used to give the location of the standard C library header files. Any directories of header files local to a specific programming project are often candidates for the "i" option method. See the "WATCOM C/386 #include File Processing" section of the "WATCOM C/386 Compiler" chapter for more details.

C.3 What do these undefined references mean?

The WATCOM Linker builds an executable file by a process of resolving undefined references. References are caused by programs referencing functions or data items that are declared in other source files. Certain conditions arise that cause the linker to generate an "undefined reference" error message. An "undefined reference" error message will be displayed by the linker when a function or data item cannot be found. The message may be a result of not indicating to the linker the correct object files and libraries to use. There are a couple of "undefined references" that require some explanation.

cstart The unresolved reference for `_cstart_` indicates that the linker cannot find the C run-time libraries. For instance, the C run-time libraries for the flat memory model compiled for the register-based argument passing model are `clib3r.lib` and `math3r.lib`. Insure that the **LIB** environment variable is set to the directory that contains the WATCOM C/386 run-time libraries. If you do not want to use the **LIB** environment variable then include the following lines in your linker command file.

```
OPTION nodefault
LIBRARY c:\penpoint\sdk\lib\math3r.lib
LIBRARY c:\penpoint\sdk\lib\penpoint.lib
```

fltused The `_fltused_` undefined reference indicates that floating-point arithmetic has been used in the modules that exhibit this error. The remedy is to ensure that the linker can find the appropriate math library. For the register-based argument passing model, it is `math3r.lib`. The **LIB** environment variable can be set to the directory containing the WATCOM C/386 libraries. The following lines can be added to the linker command file if you do not want to set the **LIB** environment variable.

```
OPTION nodefault
LIBRARY c:\penpoint\sdk\lib\math3r.lib
    and one of
LIBRARY c:\penpoint\sdk\lib\dos\clib3r.lib
    or
LIBRARY c:\penpoint\sdk\lib\win\clib3r.lib
```

main_ All C programs, except applications developed specifically for Windows, must have a function called "main". The name "main" must be in lower case for the compiler to generate the appropriate information in the "main" module.

WINMAIN All Windows programs must have a function called "WinMain". The function "WinMain" must be declared "PASCAL" in order that the compiler generate the appropriate name in the "WinMain" module.

C.4 Why aren't my variables set to zero?

The linker is the program that handles the organization of code and data and builds the executable file. Recall that C guarantees that all global and static uninitialized data will contain zeros. The "BSS" region contains all uninitialized global and static data for C programs (the name "BSS" is a remnant of the early UNIX C compilers). Most C compilers take advantage of this situation by not explicitly storing all the zeros to achieve smaller executable file sizes. In order for the program to work correctly there must be some code (that will be executed before "main") that will clear the "BSS" region. The code that is executed before "main" is called "startup" code. The linker must indicate to the startup code where the "BSS" region is located. In order to do this, the WATCOM Linker (WLINK) treats the "BSS" segment (region) in a special manner. The special variables '_edata' and '_end' are constructed by the WATCOM Linker so that the startup code knows the beginning and end of the "BSS" region.

Some users may prefer to use the linker provided by an older IBM PC and PS/2 compiler for development. In order to have the program execute correctly, some extra care must be taken with other linkers. For instance, with the Microsoft linker (LINK) you must ensure that the '/DOSSEG' command line option is used. With the Phar Lap Linker, you must use the "-DOSORDER" command line option. In general, if you must use other linkers, extract the module "CSTART" from `clib?.lib` (? will change depending on the memory model) and specify the object file `cstartobj` as the first object file to be processed by the linker. The object file `cstartobj` will contain the information necessary for the linker to build the executable file correctly.

C.5 How can I see my source files in the debugger?

The selection and use of debugging information is important for getting the most out of the WATCOM C/386 debugger, VIDEO. There are three areas where things can go wrong, namely:

1. Using the correct option for the compiler (WATCOM C/386)
2. Using the correct directives for the linker (WATCOM Linker)
3. Using the right commands in the debugger (VIDEO)

The WATCOM C/386 compiler takes C source and creates an object file containing the generated code. The compiler will output debugging information into the object file. There are two forms of debugging information that the compiler can output into the object file.

1. Line numbers and local variables ("d2" option)
2. Line numbers ("d1" option)

The particular options indicate what you will be able to see while you are debugging the particular module. If you use the "d2" option, you will be able to see your source file and display your local variables. On the other hand, the "d1" option will allow you to see your source file while you are debugging. The reason that there are two levels of debugging information available is that the code optimizer can perform many more optimizations and still maintain "d1" (line) information. The "d2" option forces the code optimizer to ensure that any local variable can be displayed at any time in the function. To illustrate why this results in less optimum code being generated for a function, let us look at a simple array initialization.

```
extern int a[100];

void init_a( void )
{
    int i;

    for( i = 0; i < 100; ++i ) {
        a[i] = 3*i;
    }
}
```

The code optimizer will ensure that you can print the value of the variable "i" at any time during the execution of the loop. The "d2" option will always generate code and debugging information so that you can print the value of any variable during the execution of the function. In order to get the best code possible and still see your source file while debugging, the "d1" option only generates line number information into the object file. With line number

information, much better code can be generated. Here is the C equivalent of the code generated for the array initialization example.

```
extern int a[100];

void init_a( void )
{
    int *t1;
    int t2;

    /* for( i = 0; i < 100; ++i ) { */
    t1 = a;
    t2 = 0;
    do {
        /* a[i] = 3*i; */
        *t1 = t2;
        ++t1;
        t2 += 3;
    /* } */
    } while( t1 != a + 100 );
}
```

The above code executes very quickly but notice that the variable "i" has been split into two different variables. One of the variables handles the use of "i" as an array index and the other handles the calculation of "3*i". The debugging of programs that have undergone extensive optimization can be difficult, but with the source line information it is much easier. To summarize, use the "d2" compiler option if you are developing a module and you would like to be able to display each local variable. The "d1" compiler option will give you line number information and the best generated code possible. There is absolutely no reason not to specify the "d1" option because the code quality will be identical to code generated without the "d1" option.

The WATCOM Linker (WLINK) is the tool that puts together a complete program and sets up the debugging information for all the modules in the executable file. There is a linker directive that indicates to the linker when it should include debugging information from the modules. There are five levels of debugging information that can be collected during the link. These are:

1. global names (DEBUG)
2. global names, line numbers (DEBUG LINE)
3. global names, types (DEBUG TYPES)
4. global names, local variables (DEBUG LOCALS)
5. all of the above (DEBUG ALL)

Notice that global names will always be included in any request for debugging information. The debugging options can be combined

```
DEBUG LINE, TYPES
```

with the above directive resulting in full line number and typing information being available during debugging. The directives are position dependent so you must precede any object files and libraries with the debugging directive. For instance, if the file `mylink.lnk` contained:

```
#
# invoke with: wlink @mylink
#
file main
debug line
file input, output
debug all
file process
```

then the modules `input` and `output` will have global names and source line information available during debugging. All debugging information in the module `process` will be available during debugging.

> ***Hint:*** A subtle point to debugging information is that all the modules will have global names available if any debugging directive is used. In the above example, the module `main` will have global name information even though it does not have a DEBUG directive preceding it.

It is preferable to have one DEBUG directive before any FILE and LIBRARY directives. You might wonder if this increases the size of the executable file so that it will occupy too much memory during debugging. The debugging information is loaded "on demand" by the debugger during the debugging session. A small amount of memory (20k default, selectable with the VIDEO "dynamic" command line option) is used to hold the most recently used module debugging information. In practice, this approach saves a lot of memory because most debugging information is never used. The overhead of accessing the disk for debugging information is negligible compared to accessing the source file information. In other words, you can have as much debugging information as you want included in the executable file without sacrificing memory required by the program. See the section entitled "The DEBUG Directive" in the *WATCOM Linker User's Guide* for more details.

If the previous steps have been followed, you should be well on your way to debugging your programs with source line information. There are instances where the WATCOM debugger, VIDEO, cannot find the appropriate source file even though it knows all the line numbers. The problem that has surfaced involves how the source file is associated with the debugging

information of the module. The original location of the source file is included in the debugging information for a module. The name that is included in the debugging information is the original name that was on the WATCOM C/386 command line. If the original filename is no longer valid (i.e., you have moved the executable to another directory), VIDEO must be told where to find the source files. The VIDEO command "SET SOURCE" supplies new directories to search for source files. If your source files are located in two directories, the following commands can be executed in VIDEO:

```
set source     {c:\program\c\*.c}
set source/add {c:\program\new\c\*.c}
```

The "*" character indicates where the module name will be inserted while VIDEO is searching for the source file. See the section entitled "Source File Search Patterns" under the description of the "SET" command in the *WATCOM VIDEO User's Guide* for more details.

> ***Hint:*** The source directory commands can be put into the file `profile.dbg` (which is usually located in the `binb` sub-directory of the installation directory). Any commands that should be executed every time VIDEO is invoked can be put into the `profile.dbg` file. The source directory commands and setting the default radix are common commands which are executed when VIDEO is invoked. Take a look at the other VIDEO command files in the same directory.

C.6 What does "Stack Overflow!" mean?

The memory used for local variables is allocated from the function call stack although the WATCOM compilers will often use registers for local variables. The size of the function call stack is limited at link-time and it is possible to exceed the amount of stack space during execution. The WATCOM run-time library will perform checks whenever a large amount of stack space is required by a function but it is up to the user to check stack requirements before calling a WATCOM run-time function. Compiling programs with stack checking will ensure that there is enough stack space to call a WATCOM run-time function.

There are various ways of protecting against stack overflow errors. First, one should minimize the amount of recursive functions used in an application program. This can be done by recoding recursive functions to use loops. Keep the amount of stack used in functions to a minimum by using and reusing static arrays whenever possible. These techniques will reduce the amount of stack space required but there still may be times where the default amount of stack space (2048 bytes) is insufficient. The WATCOM linker (WLINK) allows the user to set the amount of stack space at link-time through the WLINK directive "OPTION STACK=size" where size may be specified in bytes with an optional "k" suffix for kilobytes (1024 bytes).

Example:

```
option stack=5k
```

Debugging a program that reports a stack overflow error requires some interesting techniques. The overall plan is to isolate the offending function that tried to allocate stack storage. The routine in the WATCOM run-time library that performs the stack check is called "__STK". The strange name indicates that this is a compiler generated name and its effect is usually not visible to the user. The first thing we do is start up a debugging session.

```
C>wvideo myprog
```

In VIDEO, type:

```
DBG>new
DBG>cntbrks __STK
DBG>go
```

These commands restart the program from the first instruction, setup a breakpoint that will count the number of times the "__STK" routine is called, and start execution of the program. The program will execute with messages being printed out similar to:

```
cnt$0006 = 1
cnt$0006 = 2

cnt$0006 = 19
cnt$0006 = 20
Stack Overflow!
```

Record the last value of the count variable and execute the following VIDEO commands.

```
DBG>break/clear 1
DBG>new
DBG>bcnt 0n20 __STK
DBG>go
```

The "0n20" means 20 in decimal notation so that if the default radix is not 10, we will still get correct results. Once the debugger halts at the first instruction in __STK, the next command depends on the memory model of the program. If you are executing in a small code memory model (this includes tiny and flat memory models), type:

```
DBG>examine/source *dbg$sp
```

If you are executing in a large code memory model, type:

```
DBG>examine/source %dbg$sp
```

The function that caused the stack overflow error should be visible at this point. The solution to the stack overflow problem at this point depends on the programmer.

C.7 Why do I get redefinition errors from WLINK?

This question comes up often in discussions about porting old UNIX or Microsoft C programs. The problem stems from the forgiving nature of early UNIX linkers. In early C code, it was common to define header files like this:

Example:

```
/* define global variables */
int line_count;
int word_count;
int char_count;
```

The header file would then be included in many different modules. The C compiler would generate a definition of each variable in each module and leave it to the linker to pick one and resolve all references to one variable. The development of the ANSI C standard made this practice non-conforming. The WATCOM C compiler is an ANSI C compiler and as such, is not required to support this obsolete behaviour. The effect is that WLINK will report redefinition errors. The header file must be coded in such a way that the variables are defined in one module. One way to do this is as follows:

Example:

```
#ifdef DEFINE_HERE
#define GLOBAL
#else
#define GLOBAL  extern
#endif
/* define global variables */
GLOBAL int line_count;
GLOBAL int word_count;
GLOBAL int char_count;
```

In most modules, the macro "DEFINE_HERE" will not be defined so the file will be equivalent to:

Example:

```
/* define global variables */
extern int line_count;
extern int word_count;
extern int char_count;
```

In one module, the macro "DEFINE_HERE" must be defined before the header file is included. This can be done by defining the macro on the command line or by coding like this:

Example:

```
#define DEFINE_HERE
#include "globals.h"
```

C.8 How do I convert to WATCOM C/386?

There are some common steps involved in converting C programs written for other compilers. Conversion from UNIX and IBM PC and PS/2 DOS compilers will be covered in detail later. There are three major problems with most programs that are ported to WATCOM C/386. The assumptions that most foreign programs make that may be invalid when using WATCOM C/386 are:

1. sizeof(pointer) == sizeof(int)

   (true for 16-bit systems except "far" pointers, true for 32-bit systems except "far" pointers)

2. sizeof(long) == sizeof(int)

   (not true for 16-bit systems)

3. sizeof(short) == sizeof(int)

   (not true for 32-bit systems)

4. arguments are always passed on the stack

5. dereferencing the `NULL` pointer

6. "char" is either signed or unsigned

These assumptions are very easy to make when developing programs for only one system. The first point, becomes important when you move a program to 80x86 systems. Depending on the memory model, the size of an integer might not equal the size of a pointer. You might ask how this assumption is made in programs. The C language will assume that a function returns an integer unless told otherwise. If a programmer does not declare a function as returning a pointer, the compiler will generate code which would convert an integer to a pointer. On other systems, where the size of an integer is equal to the size of a pointer this would amount to nothing because no conversion was necessary (to change size). The older C compilers did not worry about warning the programmer about this condition and as such this error is imbedded in a lot of older C code. As C was moved to other machines, it became apparent that this assumption was no longer valid for all machines. The 80x86 architecture can have 16-bit integers and 32-bit pointers (in the compact, large, and huge memory models), which means that more care must be taken when working with declarations (converting an int to a 32-bit pointer will result in a segment value of 0x0000 or 0xffff). Similarly, the 80386 architecture can have 32-bit integers and 48-bit pointers. The WATCOM C/386 compiler will complain about incorrect pointer and integer mixing thus making programs compiled with

WATCOM C/386 much more portable. For instance, if the WATCOM C/386 compiler complains about your usage of the "malloc" memory allocation function then you probably forgot to include "<stdlib.h>" which contains the prototype of the "malloc" function.

Example:

```
extern void *malloc( unsigned );
```

The WATCOM C/386 compiler was complaining about you trying to assign an integer (the value returned by "malloc") to a pointer. By including the header file with the correct prototype, the WATCOM C/386 compiler can validate that you are in fact assigning a pointer value to a pointer.

Passing arguments on the stack has been the method used by most older compilers because it allowed the C library function "printf" to work with a variable number of arguments. Older C compilers catered to a few functions by forcing all the argument handling to be handled by the caller of the function. With the advent of the ANSI standard, which forced all functions expecting a variable number of arguments to be declared properly, compilers can generate smaller code for routines that did not require a variable number of arguments.

Example:

```
/* function accepting two arguments */
extern FILE *fopen( char *, char * );
/* function accepting a variable number of arguments */
extern int printf( char *, ... );
```

The WATCOM C/386 compiler takes advantage of this part of the ANSI standard by passing arguments in registers (for the first few arguments). If there are not enough registers for all of the arguments, the rest of the arguments are passed on the stack but the routine being called is responsible for removing them from the stack. The WATCOM C/386 compiler uses this calling convention because it results in faster procedure calls and smaller code. The WATCOM C/386 calling convention carries with it a responsibility to ensure that all functions are prototyped correctly before they are used. For instance, if a procedure is called with too few arguments, the assumptions that the code generator made (while generating the code) will be invalidated. The code generator assumes that EAX and any other registers used to pass arguments will be modified by the called function. The code generator also assumes that the exact amount of arguments pushed on the stack will be removed by the function that is called. It is important to recognize this aspect of the WATCOM C/386 compiler because the program will simply not work unless the caller and the function (being called) strictly agree on the number and types of the arguments being passed. See the "Assembly Language Considerations" chapter for more details.

Some compilers allow the `NULL` pointer to be dereferenced and return `NULL` (we have never understood the rationale behind this, nor why some compilers continue to support this type of code). Leaving the aesthetics of this type of code behind, using the `NULL` dereferencing

assumption in a program will ensure that the program will not be portable. Source code which contains the NULL dereferencing assumption must be corrected before it will work with WATCOM C/386.

Programs that assume that the "char" type is "signed" should use the WATCOM C/386 compiler "j" option. The "j" option will indicate to the WATCOM C/386 compiler that the "char" type is "signed" rather than the default "unsigned".

C.8.1 Conversion from UNIX compilers

The ANSI standard for C (which WATCOM C/386 adheres to) is very similar to UNIX C. Most of the effort in converting UNIX C programs will involve replacing references to library functions (such as the CURSES library). There are many third-party libraries which are implementations of UNIX libraries on the IBM PC and PS/2. There is a common problem which many older UNIX programs exhibit, namely, functions that accept a variable number of arguments are coded in many different ways. Functions accepting a variable number of arguments must be coded according to the ANSI standard if they are to work with WATCOM C/386. We will code an example of a function which will return the maximum of a list of positive integers.

```
/*
  variable number of arguments example
*/
#include <stdarg.h>

int MaxList( int how_many, ... )
{
    va_list args;
    int max;

    max = 0;
    va_start( args, how_many );
    while( how_many > 0 ) {
        value = va_arg( args, int );
        if( value > max ) {
            max = value;
        }
    }
    va_end( args );

    return( max );
}
```

Notice that the standard header file stdarg.h must be included in any source file which defines a function that handles a variable number of arguments. The function "MaxList" must

be prototyped correctly in other source files external to the source file containing the definition of "MaxList".

```
extern int MaxList( int how_many, ... );
```

See the *WATCOM C Library Reference* manual description of "va_arg" for a more complete description of variable number of arguments handling.

C.8.2 Conversion from IBM PC and PS/2 DOS compilers

Most of the compilers available for the IBM PC and PS/2 have been following the recently approved ANSI standard and as such, the majority of programs will not require extensive source changes. There are problems with programs that use compiler-specific library functions. The use of compiler-specific library functions can be dealt with in two different ways:

1. use equivalent WATCOM C/386 library functions
2. write your own library functions

If portability must be maintained with the other compiler, the predefined macro "`__WATCOMC__`" can be used to conditionally compile the correct code for the WATCOM C/386 compiler.

The default calling convention for the WATCOM C/386 compiler is different from the calling convention used by other IBM PC and PS/2 compilers. The WATCOM C/386 calling convention is different because it will pass some arguments in registers (thus reducing the overhead of a function call) rather than pushing all of the arguments on the stack. The WATCOM C/386 compiler is flexible enough to use different calling conventions on a per function basis. Converting code from other compilers usually involves recompiling the C source files and setting up prototypes (to use the older calling convention) for functions written in assembly language. For instance, if you have the functions "video_init", "video_put", and "video_get" written in assembly language, you can use the following prototypes in any source file which uses these functions.

```
#include <stddef.h>

extern int  cdecl video_init( void );
extern void cdecl video_put( int row,int col,char ch,int attr );
extern char cdecl video_get( int row,int col );
```

The inclusion of the `stddef.h` header file defines the "cdecl" calling convention. The WATCOM C/386 compiler will ensure that any calls to these three functions will adhere to the "cdecl" calling conventions. The WATCOM C/386 compiler will put a trailing underscore "_" character (as opposed to the beginning of the name for the "cdecl" convention) on any

function names to ensure that the program will not link register calling convention calls to "cdecl" convention functions (or vice versa). If the linker indicates that functions defined in assembler files cannot be resolved, it could be a result of not prototyping the functions properly as "cdecl" functions.

C.9 What should I know about optimization?

The C language contains features which allow simpler compilers to generate code of reasonable quality. Register declarations and imbedding assignments in expressions are two of the ways that C allows the programmer to "help" the compiler generate good quality code. An important point about the WATCOM C/386 compiler is that it is not as important (as it is with other compilers) to "help" the compiler. In order to make good decisions about code generation the WATCOM C/386 compiler uses modern optimization techniques.

> ***Hint:*** The definitive reference on compiler design is the "dragon" book "Compilers - Principles, Techniques, and Tools", Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, published by Addison-Wesley, Reading, Massachusetts, 1986. The authors of the "dragon" book advocate a conservative approach to code generation where optimizations must preserve the semantics of the original program. The conservative approach is used throughout the WATCOM C/386 compiler to ensure that programmers can use the compiler without worrying about the semantics of their program being changed. The programmer can request that potentially unsafe optimizations be performed. With regard to the "oa" (ignore aliasing) option provided by the WATCOM C/386 compiler, the compiler only ignores aliasing of global variables rather than ignore aliasing totally like other compilers.

There are certain pieces of information which the compiler cannot derive from the source code. The "#pragma" compiler directive is used to provide extra information to the compiler. It is necessary to have a complete understanding of both C and the machine architecture (i.e., 80x86) before using the powerful "#pragma" compiler directives. See the "Pragmas" chapter for more details.

Debugging optimized programs is difficult because variables can be assigned to different locations (i.e., memory or registers) in different parts of the function. The "d2" compiler option will restrict the amount of optimization so that variables occupy one location and can be easily displayed. It follows that the "d2" option is useful for initial development but production programs should be compiled with only the "d1" option for the best code quality.

> ***Hint:*** Recall that the "d2" compiler option will generate symbolic information (for every local variable) and line number information for the source file. The "d1" compiler option will only generate line number information for the source file. The use of these options determines what kind of information will be available for the particular module during the debugging session.

Incorrect programs can sometimes work when compiled with the "d2" option and not work when compiled with the "d1" option. One way this sort of problem arises involves local arrays.

```
void example( void )
{
    int i;
    int a[10];

    for( i = 0; i <= 10; ++i )
        a[i] = i;
    do_something( a );
}
```

The "for" loop initializes one too many array elements but the version compiled with the "d2" option will overwrite the variable "i" without causing any problems. The same function compiled with the "d1" option would have the variable "i" in a register. The erroneous access of "a[10]" would modify a value that is used to restore a register when the function returns. The register would be "restored" with an incorrect value and this would affect the execution of the function that called this function. The above example shows how a program can work when compiled with the "d2" option and stop working when compiled with the "d1" option. You should always test your program fully with all the modules compiled with the "d1" option to protect yourself from any surprises.

D. Software Problem Reporting

Although a significant amount of testing has gone into this product, errors in the software may be encountered. You also may discover errors or omissions in the documentation. If you find any problems or have some good suggestions, we would like to hear from you.

PenPoint developers should contact GO Corporation for technical support relating to the PenPoint operating system. If you have a problem regarding the compiler and related tools (Linker, Disassembler, etc.), you should contact WATCOM for technical support.

The form on the following page may be used when reporting a problem or making a suggestion to WATCOM by mail or facsimile transmission. When reporting a problem, please retain a copy of your completed form for your own records.

Return your completed report to:

```
WATCOM                          Telephone: (519) 886-3700
415 Phillip Street              FAX:       (519) 747-4971
Waterloo, Ontario               BBS:       (519) 884-2103
CANADA   N2L 3X2
```

If you wish, you can access our bulletin board service. You can read about known problems and down-load available fixes for the latest version of the software. You can also report problems using this system. The bulletin board is capable of communications at 300, 1200, 2400, 4800, 7200, 9600, 12000, and 14400 BAUD. For BAUD rates higher than 2400, a Courier HST modem from U.S. Robotics or compatible V.32 modem is required. The word length is 8, the parity setting is "none", and stop bits must be 1.

If you wish to report a problem to us by phone, you may use the WATCOM technical support hot line. The phone number is (519) 884-0702.

Software Problem Report

Program Name/Version:	Date:
Computer Model:	Memory Size:
Processor Type:	Coprocessor:
Monitor and Adapter Types:	
Operating System and Version:	
Your Name: Institution: Address:	Telephone: FAX Number:
Please provide a description of problem and how it can be reproduced. Attach listings, console output, and anything else that you may consider relevant.	

For WATCOM Use Only

Received by:	Date:
Resolution:	

#

#include 42
#line 29

3

386LINK 316

\

\INC directory 42

_

__386__ 46
__based 48
__based keyword 48
__cdecl 48-49
__cdecl keyword 48
__CHAR_SIGNED__ 20, 47
__COMPACT__ 21, 47
__export 49-50
__export keyword 49
__EXPRESSC__ 47
__far 48
__far keyword 48
__far16 51
__far16 keyword 51
__FLAT__ 20, 47
__fortran 49
__FPI__ 19, 47
__GRO 30
stack growing 30
__huge 48
__huge keyword 48
__INLINE_FUNCTIONS__ 27, 47
__interrupt 49
__interrupt keyword 49
__LARGE__ 21, 47
__loadds 50
__loadds keyword 50
__MEDIUM__ 21, 47
__near 48
__near keyword 48
__NETWARE_386__ 46
__OS2__ 45
__pascal 49
__pascal keyword 49
__QNX__ 45
__saveregs 50
__saveregs keyword 50
__segment 48
__segment keyword 48
__segname 48
__segname keyword 48
__self 48
__self keyword 48
__SMALL__ 20, 47
__STK 30
stack overflow 30
__syscall 50
__syscall keyword 50
__WATCOMC__ 46, 328
__WINDOWS_386__ 37, 46
__WINDOWS__ 46
_BSS segment 21
_cdecl 49
_cdecl keyword 34, 48
cstart 315
_DATA segment 21
_edata linker symbol 86
_end linker symbol 86

_export 50
_export keyword 34, 49
_far 48
_far keyword 34, 48
fltused 315
_fortran 49
_fortran keyword 34
_huge 48
_huge keyword 34, 48
_interrupt 49
_interrupt keyword 34, 49
_loadds 50
_loadds keyword 34, 50
_near 48
_near keyword 34, 48
_pascal 49
_pascal keyword 34, 49
_saveregs 50
_saveregs keyword 34, 50
_Seg16 51
_Seg16 keyword 51
_TEXT segment 21, 23

A

ABORTS 262
  pragmas 262
addressing base 234
addressing base + (index * scale) + displacement 235
addressing base + displacement 234
addressing base + index + displacement 235
addressing direct 233-234
addressing immediate 234
addressing index * scale + displacement 234
addressing indirect 233
AFTER 196
  WMAKE directive 196
alias names 243
  in pragmas 243
aliasing 24
ALIGNMENT option for PenPoint executable files 84
alloc_text 242
alloc_text pragma 242
ANSI compatibility 31
ANSI standard 326
  variable number of arguments 326
argument lists 252
  in pragmas 252
arguments 257, 326
  removing from the stack 257
  what you need to know 326
arguments on the stack 255
array subscript errors 331
  how they hurt 331
auto 296-297, 300
AUTOEXEC.BAT 9, 38, 314
  system initialization file 9, 38, 314
auxiliary carry flag 231
auxiliary pragmas 242

B

back end 51
BBS 333
BEFORE 196
  WMAKE directive 196
Bell Laboratories 155
BIN directory 280
BINB directory 280
BINP directory 280
BLOCK 149
  WMAKE directive 149
BPATCH 219
  command line format 219
break 290
BSS class 21
BSS segment 316
bugs 219

bulletin board service 333

C

calling convention 29, 245, 269, 326
  MetaWare High C 29, 245, 269
  what you need to know 326
calling conventions 328
  cdecl 328
calling functions 249
  far 249
  near 249
calling information 249
  pragmas 249
carry flag 231
case 288, 290, 299
CASEEXACT option 84
cdecl 49, 328
  calling convention 328
cdecl keyword 34, 48, 246
char 20, 294, 296
check_stack option 239
checking macro values 193
class 21
  BSS 21
  CODE 21
  DATA 21
clearing 316
  variables 316
CMD.EXE shell 198
CODE class 21
code generation 51-52, 282
  memory requirements 52, 282
code segment 31
colon (:) 155, 158
  behaviour in WMAKE 158
  explicit rule in WMAKE 155
command execution 198
command line format 5, 13, 57, 148, 207, 209, 219
  BPATCH 219
  WCC386 13
  WCL386 5
  WDISASM 209
  WLINK 57
  WMAKE 148
  WTOUCH 207
command line format 127, 143
  WLIB 127
  WSTRIP 143
COMMAND.COM shell 198
comment (#) directive 102
common information 187
communication 200
compiling command line format 13
compiling options 14, 16
CONFIG.SYS 9, 38
  system initialization file 9, 38
const 291, 296
CONST segment 21
CONST2 segment 21
CONTINUE 150, 290
  WMAKE directive 150
conversion 325
  problems 325
  what you need to know 325
converting Microsoft response files to WATCOM Linker directive files 106
converting to WATCOM C/386 325, 327-328
  common problems 325
  from IBM PC and PS/2 compilers 328
  from UNIX 327
  what you need to know 325
CS register 228-229

D

DATA class 21
DEBUG directive 64
debug information 143

removal 143
DEBUG options 64
ALL 64
LINES 64
LOCALS 64
TYPES 64
debugging 317, 321, 330
optimized programs 317, 330
stack overflow 321
techniques 321
debugging information 64-68, 317-319
all 66
global symbol 64, 67
global variables 317
line numbering 64-65, 317
local symbol 64-65
local variables 317
source file 317
strip from "EXE" file 68
types 317
typing 64, 66
VIDEO 319
WCC386 317
what can go wrong 317
WLINK 318
debugging makefiles 149, 202
declarations 155
DEF files 186
DEFAULT 196, 290-291, 299
WMAKE directive 196
default directive file 63, 102
default libraries 240
using pragmas 240
dependency 155
dependent 155
dependent extension 174
DESCRIPTION option for PenPoint executable files 85
DGROUP group 21
diagnostics 40-41, 307, 309-310
errno 309
error 41
informational 307
matherr 310
run-time 309-310
warning 41
WATCOM C/386 40
diagnostics error 290
diagnostics warning 286, 289
different memory model libraries 187
direct addressing 233-234
direction flag 232
directives 63-64, 69, 71, 75, 78-82, 97, 100, 102-103
comment 102
DEBUG 64
FILE 69
FORMAT 71
include 103
LIBFILE 78
LIBPATH 79
LIBRARY 75
MODTRACE 80
NAME 81
NEWSEGMENT 82
PATH 97
SYMTRACE 100
SYSTEM 100
directives for PenPoint executable files 68, 74, 82, 98
EXPORT 68
IMPORT 74
OPTION 82
SEGMENT 98
disassembler 209
DLL 21, 37, 49-50
exporting functions 49
do 290, 299
DOS or OS/2 batch files 189
DOSSEG option 85
double 296, 300
double colon explicit rule 185
double-colon (::) 185-186
behaviour in WMAKE 186
explicit rule in WMAKE 185
dragon book 330
DS register 228-229
DS segment register 49-50

duplicated information 187
dynamic link library 21, 37, 49-50
  exporting functions 49
dynamic variables 200

E

EAX register 227
EBP register 227
EBX register 227
echo 196
  WMAKE 196
ECX register 227
EDI register 228
EDX register 227
effective address 233
EIP register 229
EIP relative instruction fetch 233
enum 292, 302, 305
environment string 9, 38
  # 9, 38
  = substitute 9, 38
environment variables 9, 11, 37-38, 42-44, 51-53, 76-77, 102, 106, 168, 191-192, 199-200, 219, 279-283, 314-315
  FORCE 279
  INCLUDE 42-44, 199-200, 279, 314
  LIB 76, 192, 199, 279, 315
  LIBDOS 77
  LIBNOV 77
  LIBOS2 77
  LIBPHAR 77
  LIBQNX 77
  LIBWIN 77
  PATH 43, 51, 102, 168, 219, 279-280
  TMP 106, 280-281
  WATCOM 281
  WCC386 37-38, 281
  WCGMEMORY 52-53, 282-283
  WCL386 9, 11, 38, 281-282
ERASE 150, 163
  WMAKE directive 150, 163
errno 309-310
  E2BIG 309
  EACCES 309
  EBADF 309
  EDEADLK 310
  EDOM 310
  EEXIST 309
  EINVAL 309
  EMFILE 310
  ENFILE 309
  ENOENT 309
  ENOEXEC 309
  ENOMEM 309
  ENOSPC 310
  ERANGE 310
  EXDEV 309
ERROR 197
  WMAKE directive 197
error codes 309-310
  ERRNO.H 309
  MATH.H 310
error file 40
  .err 40
error messages 290
ES register 228-229
ESI register 228
ESP register 227
example 327
  variable number of arguments 327
executable files 143
  reducing size 143
explicit rule 155, 185
EXPORT directive for PenPoint executable files 68
EXTENSIONS 175, 186
  WMAKE directive 175, 186
extern 293, 298, 302

F

facsimile transmission 333
far 23, 37, 40, 48, 249
  pragmas 249
far keyword 34, 48
FAX 333
Feldman, S.I. 155
FILE directive 69
files 314
  unable to find 314
finding targets 178
flags 229
flags auxiliary carry 231
flags carry 231
flags direction 232
flags i/o privilege level 231
flags interrupt enable 232
flags nested task 231
flags overflow 233
flags parity 231
flags resume 231
flags sign 232
flags trap 232
flags virtual 8086 mode 231
flags zero 232
float 296, 300
floating-point 19
  consistency of options 19
  option 19
FOR 201, 290, 301
  using WATCOM Make 201
FORCE environment variable 279
FORMAT directive 71
fortran 49, 287
fortran keyword 34
FRAME 251
  pragmas 251
front end 51
FS register 228-229

G

general registers 225
generating prototypes 186
global recompile 149, 189
goto 288, 292, 294
group 21
  DGROUP 21
GS register 228-229

H

header file 42
  including 42
  searching 42
header files 314
HEAPSIZE option for OS/2 executable files 86
HEAPSIZE option for PenPoint executable files 86
help! 314-315
  header files 314
  unable to find files 314
  undefined references 315
  weird undefined references 315
HOLD 152, 163
  WMAKE directive 152, 163
huge 48
huge keyword 34, 48

I

i/o privilege level 231
IBM PC and PS/2 compilers 328

IGNORE 150, 162
WMAKE directive 150, 162
ignoring return codes 162
immediate addressing 234
implicit rule 174
implicit rules 176
$[form 176
$] form 176
$^ form 176
import definitions 61
IMPORT directive for PenPoint executable files 74
import library 62
in-line assembly language instructions 249-250
in pragmas 249
using mnemonics 250
in-line functions 250, 256
pragmas 256
include 42
directive 42
header file 42
source file 42
include directive 103
INCLUDE environment variable 42-44, 200, 279, 314
include file 42
searching 42
indirect addressing 233
initialization file 198
initialized global data 316
instruction fetch 233
EIP relative 233
instruction fetching 233
int 294, 296, 304
integer/pointer equivalence 325
interrupt 49
interrupt enable flag 232
interrupt keyword 34, 49
interrupt routine 49
INTPACK union 49
invalid conversion 325
invoking WATCOM C/386 13
invoking WATCOM Linker 57
invoking WATCOM Make 155, 188, 190
invoking WATCOM Touch 207
IP register 233

K

keywords 34, 48-51, 246
__based 48
__cdecl 48
__export 49
__far 48
__far16 51
__fortran 49
__huge 48
__interrupt 49
__loadds 50
__near 48
__pascal 49
__saveregs 50
__segment 48
__segname 48
__self 48
__syscall 50
_cdecl 34, 48
_export 34, 49
_far 34, 48
_fortran 34, 49
_huge 34, 48
_interrupt 34, 49
_loadds 34, 50
_near 34, 48
_pascal 34, 49
_saveregs 34, 50
_Seg16 51
cdecl 34, 48, 246
far 34, 48
fortran 34, 49
huge 34, 48
interrupt 34, 49
near 34, 48
pascal 34, 49, 246

syscall 246
system 246

L

large projects 187
larger applications 180
LIB environment variable 76, 279, 315
LIBDOS environment variable 77
LIBFILE directive 78
LIBNOV environment variable 77
LIBOS2 environment variable 77
LIBPATH directive 79
LIBPHAR environment variable 77
LIBQNX environment variable 77
libraries 187
library file 129-131
  adding to a 129
  deleting from a 130
  extracting from a 131
  replacing a module in a 130
LIBRARY linker directive 75
library manager 127
library path 281
LIBWIN environment variable 77
line 29
  directive 29
line continuation 167
line number information 330
LINK 316
LINK386 316
linker 315
  undefined references 315
  weird undefined references 315
linker symbols 86
  _edata 86
  _end 86
linking notation 57
LOADDS 250
  pragmas 250
loading DS before calling a function 250
loading DS in prologue sequence of a function 251
long 296

M

M_I86 46
M_I86CM 21, 47
M_I86LM 21, 47
M_I86MM 21, 47
M_I86SM 20, 47
macro construction 169, 186
macro definition 193
macro identifier 191
macro text 193
macros 19-21, 27, 31, 37, 45-47, 164, 193, 328
  __386__ 46
  __CHAR_SIGNED__ 20, 47
  __COMPACT__ 21, 47
  __EXPRESSC__ 47
  __FLAT__ 20, 47
  __FPI__ 19, 47
  __INLINE_FUNCTIONS__ 27, 47
  __LARGE__ 21, 47
  __MEDIUM__ 21, 47
  __NETWARE_386__ 46
  __OS2__ 45
  __QNX__ 45
  __SMALL__ 20, 47
  __WATCOMC__ 46, 328
  __WINDOWS_386__ 37, 46
  __WINDOWS__ 46
  M_I86 46
  M_I86CM 21, 47
  M_I86LM 21, 47
  M_I86MM 21, 47
  M_I86SM 20, 47
  MSDOS 45
  NO_EXT_KEYS 31, 47

main_ 315
maintaining libraries 187
maintenance 147
make 147, 188, 207
  include file 188
  Touch 207
  WMAKE 147
MAKEFILE 150, 155
MAKEFILE comments 155
MAKEINIT 198
MANYAUTODATA option for PenPoint executable files 87
map file 87
MAP option 87
math coprocessor 19
  option 19
matherr 310-311
  DOMAIN 310
  OVERFLOW 310
  PLOSS 310
  SING 311
  TLOSS 311
  UNDERFLOW 311
memory layout 85, 105
memory model 38, 187
memory models 325
  what you need to know 325
memory requirements 106
message passing 200
messages 307, 309-310
  errno 309
  informational 307
  matherr 310
  run-time 309-310
messages error 290
messages warning 286, 289
MetaWare 29, 245, 269
  High C calling convention 29, 245, 269
Microsoft 316
  LINK 316
  LINK386 316
modem 333
  Courier HST 333
modification 207
MODIFY EXACT 268-269
  pragmas 268-269
MODIFY NOMEMORY 263, 266
  pragmas 263, 266
MODIFY {reg_set} 275
  pragmas 275
MODNAME option for PenPoint executable files 88
MODTRACE directive 80
MS2WLINK command 106
MSDOS 45
multiple dependents 156
multiple source directories 180
multiple targets 156

N

NAME directive 81
NAMELEN option 88
near 48, 249
  pragmas 249
near keyword 34, 48
nested task flag 231
NEWFILES option for PenPoint executable files 89
NEWSEGMENT directive 82
NO8087 258
  pragmas 258
NO_EXT_KEYS 31, 47
NOAUTODATA option for PenPoint executable files 89
NOCHECK 149, 158, 205
  WMAKE directive 149, 158, 205
NODEFAULTLIBS option 89
notation 57
NULL pointer 326
numeric data processor 19
  option 19

O

OFFSET option for PenPoint executable files 90
OLDLIBRARY option for PenPoint executable files 91
ONEAUTODATA option for PenPoint executable files 91
optimization 24, 330
  suggested reading 330
  what you should know 330
OPTIMIZE 151, 182
  WMAKE directive 151, 182
OPTION directive for PenPoint executable files 82
options 84-85, 87-89, 93-96, 239
  CASEEXACT 84
  check_stack 239
  DOSSEG 85
  MAP 87
  NAMELEN 88
  NODEFAULTLIBS 89
  QUIET 93
  SYMFILE 94
  UNDEFSOK 95
  unreferenced 239
  using pragmas 239
  VERBOSE 96
options for OS/2 executable files 86
  HEAPSIZE 86
options for PenPoint executable files 84-92, 94, 96
  ALIGNMENT 84
  DESCRIPTION 85
  HEAPSIZE 86
  MANYAUTODATA 87
  MODNAME 88
  NEWFILES 89
  NOAUTODATA 89
  OFFSET 90
  OLDLIBRARY 91
  ONEAUTODATA 91
  PROTMODE 92
  STUB 94
  VERSION 96
overflow flag 233

P

pack pragmas 241
PACKCODE linker option for DOS executable files 91
parameters 326
  what you need to know 326
parity flag 231
PARM 252
  pragmas 252
PARM CALLER 257
  pragmas 257
PARM NOMEMORY 266
  pragmas 266
PARM REVERSE 257
  pragmas 257
PARM ROUTINE 257
  pragmas 257
PARM {reg_set} 270
  pragmas 270
pascal 49
pascal keyword 34, 49, 246
passing arguments 271
  in 80x87 registers 271
patches 219
path 178
PATH directive 97
PATH environment variable 43, 51, 102, 168, 219, 279-280
pause 196
  WMAKE 196
PenPoint dynamic link libraries 61
PenPoint program modules 61

Phar Lap 316
386LINK 316
portability 326-327
NULL pointer 326
signed char 327
porting 323
from Microsoft C 323
from UNIX 323
pragma 237, 330
pragma options 239
pragmas 237-238, 240, 242, 245, 248-252, 256-263, 266, 268-270, 274-275
= {constant} 249
ABORTS 262
alias names 245
alternate names 248
auxiliary 242
calling information 249
describing argument lists 252
describing return value 258
FAR 249
FRAME 251
in-line assembly language instructions 249
in-line functions 256
LOADDS 250
MODIFY EXACT 268-269
MODIFY NOMEMORY 263, 266
MODIFY {reg_set} 275
NEAR 249
NO8087 258
notation used to describe 237
PARM 252
PARM CALLER 257
PARM NOMEMORY 266
PARM REVERSE 257
PARM ROUTINE 257
PARM {reg_set} 270
specifying default libraries 240
STRUCT CALLER 258, 260
STRUCT FLOAT 258, 261
STRUCT ROUTINE 258, 260
syntax 238
VALUE 258-261
VALUE NO8087 262
VALUE {reg_set} 274
pragmas pack 241
PRECIOUS 161, 163
WMAKE directive 161, 163
predefined macros 19-21, 27, 31, 37, 45-47, 328
__386__ 46
__CHAR_SIGNED__ 20, 47
__COMPACT__ 21, 47
__EXPRESSC__ 47
__FLAT__ 20, 47
__FPI__ 19, 47
__INLINE_FUNCTIONS__ 27, 47
__LARGE__ 21, 47
__MEDIUM__ 21, 47
__NETWARE_386__ 46
__OS2__ 45
__QNX__ 45
__SMALL__ 20, 47
__WATCOMC__ 46, 328
__WINDOWS_386__ 37, 46
__WINDOWS__ 46
M_I86 46
M_I86CM 21, 47
M_I86LM 21, 47
M_I86MM 21, 47
M_I86SM 20, 47
MSDOS 45
NO_EXT_KEYS 31, 47
predictable code size 52, 282
preprocessing directives 187
WMAKE 187
preprocessor restrictions 192
problems 331
with d2 and d1 options 331
program maintenance 147
PROTMODE option for PenPoint executable files 92
prototypes 186
pushing arguments 326
what you need to know 326

Q

QUIET option 93

R

recompile 149, 184, 189
reducing maintenance 190
register 292, 296-297, 302, 304
registers 225, 326
  calling convention 326
registers CS 228-229
registers DS 228-229
registers EAX 227
registers EBP 227
registers EBX 227
registers ECX 227
registers EDI 228
registers EDX 227
registers EIP 229
registers ES 228-229
registers ESI 228
registers ESP 227
registers FS 228-229
registers GS 228-229
registers SS 228-229
removing debug information 143
replace 188
resume flag 231
return 287, 295, 300-301
return codes 161, 163
return value 258
  in pragmas 258
rule command list 155
run-time 286, 309
  error messages 286, 309

S

save/restore segment registers 29
segment 21, 23
  _BSS 21
  _DATA 21
  _TEXT 21, 23
  CONST 21
  CONST2 21
SEGMENT directive for PenPoint executable files 98
segment ordering 85, 105
segment references 48
segment registers 37, 49-50
  DS 49-50
  SS 37
segname references 48
self references 48
SET 9, 11, 37, 43, 199, 201, 279-281
  FORCE environment variable 279
  INCLUDE environment variable 43, 199, 279
  LIB environment variable 199
  PATH environment variable 280
  TMP environment variable 280
  using WATCOM Make 199, 201
  WCC386 environment variable 37, 281
  WCL386 environment variable 9, 11, 281
setting 207
  modification date 207
  modification time 207
setting environment variables 192, 199
shell 198
  CMD.EXE 198
  COMMAND.COM 198
short 294, 296
side effects of functions 263
sign flag 232
signed 294, 296
signed char 327

SILENT 164
  WMAKE directive 164
single colon explicit rule 155
sizeof 302
software problems 333
software quality assurance 52, 282
source file 42
  including 42
  searching 42
SS register 228-229
SS segment register 37
STACK 93
stack frame 251
stack growing 30
STACK option 93
stack overflow 30, 321
stacking arguments 255
static 293-294, 298
struct 292-294, 300-304
STRUCT CALLER 258, 260
  pragmas 258, 260
STRUCT FLOAT 258, 261
  pragmas 258, 261
STRUCT ROUTINE 258, 260
  pragmas 258, 260
STUB option for PenPoint executable files 94
SUFFIXES 204
  WMAKE directive 204
suppressing output 164
switch 288, 290-291, 294, 301
symbol file 94
SYMBOLIC 159, 168-169, 202
  WMAKE directive 159, 168-169, 202
symbolic information 330
SYMFILE option 94
SYMTRACE directive 100
syscall keyword 246
SYSTEM directive 100
system initialization file 9, 38, 199, 314
  AUTOEXEC.BAT 9, 38, 314
  CONFIG.SYS 9, 38
system keyword 246
system name 100

T

target 155
target deletion prompt 150, 152
technical support 333
time-stamp 147, 207
TMP environment variable 106, 280-281
touch 149, 152, 196, 207
touch utility 207
trap flag 232

U

unable to find files 314
undefined references 315
  cstart 315
  fltused 315
  main_ 315
  WinMain 315
UNDEFSOK option 95
union 292-294, 300-304
UNIX 155, 204, 210, 212, 327
UNIX compatibility mode in Make 152
unreferenced option 239
unsigned 294, 296, 304
using environment variables in directives 63

V

VALUE 258-261
  pragmas 258-261
VALUE NO8087 262
  pragmas 262

VALUE {reg_set} 274
pragmas 274
variable number of arguments 327
variables 316
set to zero 316
VERBOSE option 96
VERSION option for PenPoint executable files 96
virtual 8086 mode 231
void 287, 293, 295
volatile 296

W

warning messages 286, 289
level 1 286
level 2 289
level 3 289
WATCOM C/386 51, 325-326
back end 51
calling convention 326
converting to 325
front end 51
unique aspects 325-326
WATCOM C/386 options 314, 317, 330
d1 317, 330
d2 317, 330
i 314
WATCOM environment variable 281
WATCOM Make 147
WMAKE 147
WCC386 14, 38
WCC386 command line format 13
WCC386 environment variable 37-38, 281
WCC386 options 6-7, 14-21, 23-31, 34-37, 42, 44
3r 7, 15, 28
3s 7, 15, 28
4r 7, 15, 29
4s 7, 15, 29
7 7, 15, 29
bd 6, 14, 16
bm 6, 14, 16
bw 6, 14, 16
d 6, 14, 17
d1 6, 14, 16
d1+ 6, 14, 16
d2 6, 14, 16
d2~ 6, 14, 16
e 6, 14, 17
ei 6, 14, 17
en 6, 14, 17
ez 6, 14, 17
fi 6, 14, 17
fo 6, 14, 18
fpc 6, 14, 19
fpi 6, 14, 19
fpi287 6, 14, 19
fpi87 6, 14, 19
g 6, 14, 20
i 6, 14, 20, 42, 44
j 6, 14, 20
mc 6, 14, 21
mf 6, 15, 20
ml 6, 14, 21
mm 6, 14, 21
ms 6, 14, 20
nc 6, 15, 21
nd 6, 15
nm 6, 15, 23
nt 6, 15, 21, 23
oa 7, 15, 24
oc 7, 15, 24-25
od 7, 15, 24-25
oe 7, 15, 24-25
of 7, 15, 24-25
of+ 24, 26
oi 7, 15, 24, 27
ol 7, 15, 24, 27
om 7, 15, 24
or 7, 15, 24, 27
os 7, 15, 24, 27
ot 7, 15, 24, 27
ox 7, 15, 24, 28

p 7, 15, 29
pc 7, 15, 29
pcl 7, 15, 29
pl 7, 15, 29
plc 7, 15, 29
r 7, 15, 29
s 7, 15, 30
sg 7, 15, 30
u 7, 15, 30
v 7, 15, 30
w 7, 15, 30
we 7, 15, 31
za 7, 15, 31
zc 7, 15, 31
zdf 7, 15, 31
zdp 7, 15, 31
ze 7, 15, 31
zff 7, 15, 34
zfp 7, 15, 34
zg 7, 15, 34
zk 7, 15, 34
zl 7, 15, 35
zm 7, 15, 35
zp 7, 15, 35
zq 7, 15, 36
zs 7, 15, 36
zt 7, 15, 36
zu 7, 15, 37
zW 7, 15, 37
WCC386P 14, 38
WCGMEMORY environment variable 52-53, 282-283
WCL386 5, 9-11, 305
command line format 5
WCL386 environment variable 9, 11, 38, 281-282
WCL386 options 6, 8-9
@ 9
c 6
compile 6
fd[=<directive_file>] 8
fe=<executable> 8
fm[=<map_file>] 8
k<stack_size> 8
l=<system_name> 8
p 6
x 9
y 6
WDISASM 209
command line format 209
WDISASM options 210-214
a 212
b 212
c 214
e 213
l (lowercase L) 210
p 213
r 212
s 211
u 212
weird undefined references 315
while 290, 299, 301
WinMain 315
WLIB 127, 129
command line format 127
operations 129
WLIB options 134
WLIB options 134-138
b 134
c 134
d 134
i 135
l (lower case L) 135
n 136
o 136
p 137
q 137
s 137
t 138
x 138
WLINK 57
command line format 57
WLINK command line 57
invoking WLINK 57
WLINK debug options 170
WLINK notation 57
WLINK options for DOS executable files 91
PACKCODE 91

WMAKE 147, 149-150, 152-153, 155-156, 158, 161-164, 167-169, 174, 178, 180, 184-193, 196, 198-200, 202, 204
! command execution 198
":" behaviour 158
":" explicit rule 155
"::" behaviour 186
"::" explicit rule 185
.DEF files 186
Bell Laboratories 155
checking macro values 193
command execution 198
common information 187
debugging makefiles 149, 202
declarations 155
dependency 155
dependent 155
dependent extension 174
different memory model libraries 187
DOS or OS/2 batch files 189
double colon explicit rule 185
duplicated information 187
dynamic variables 200
environment variables 168, 191-192, 199
explicit rule 155, 185
Feldman, S.I. 155
finding targets 178
ignoring return codes 162
implicit rule 174
include file 188
initialization file 198
large projects 187
larger applications 180
libraries 187
line continuation 167
macro construction 169, 186
macro definition 193
macro identifier 164, 191
macro text 193
macros 164, 193
maintaining libraries 187
MAKEFILE 150, 155
MAKEFILE comments 155
MAKEINIT 198
memory model 187
multiple dependents 156
multiple source directories 180
multiple targets 156
path 178
preprocessing directives 187
preprocessor restrictions 192
recompile 184
reducing maintenance 190
return codes 161, 163
rule command list 155
setting environment variables 192, 199
single colon explicit rule 155
special macros 153
suppressing output 164
target 155
target deletion prompt 150, 152
time-stamp 147
touch 149, 152, 196
UNIX 155, 204
UNIX compatibility mode 152
WTOUCH 196
WMAKE command line 148, 155, 188, 190-191
defining macros 148, 191
format 148
help 148
invoking WMAKE 148, 155, 188, 190
options 148
summary 148
targets 148, 190
WMAKE command prefix 162, 164
- 162
@ 164
WMAKE directives 149-152, 158-159, 161-164, 168-169, 175, 182, 186, 196-197, 202, 204-205
.AFTER 196
.BEFORE 196
.BLOCK 149
.CONTINUE 150
.DEFAULT 196
.ERASE 150, 163
.ERROR 197
.EXTENSIONS 175, 186

.HOLD 152, 163
.IGNORE 150, 162
.NOCHECK 149, 158, 205
.OPTIMIZE 151, 182
.PRECIOUS 161, 163
.SILENT 164
.SUFFIXES 204
.SYMBOLIC 159, 168-169, 202
WMAKE internal commands 202-203, 205
%abort 202-203
%append 202-203
%create 202-203
%make 202
%null 202, 205
%quit 202-203
%stop 202
%write 202
WMAKE options 149-152, 155, 162, 164, 184, 189, 191
a 149, 184, 189
b 149
c 149
d 149
e 150
f 150, 155, 191
h 150
i 150, 162
k 150
l 150
m 150
n 151
o 151
p 151
q 151
r 152
s 152, 164
t 152
u 152
z 152
WMAKE preprocessing 187, 192, 194-195
!define 194
!else 192
!endif 192
!error 195
!ifdef 192
!ifeq 192
!ifndef 192
!ifneq 192
!include 187
!undef 195
WMAKE special macros 153-154, 168-169, 171, 173, 176, 186, 191, 199, 205
$# 153, 173
$$ 153, 173
$(%<environment_var>) 168, 191
$(%cdrive) 168
$(%cwd) 168
$(%path) 168, 199
$* 153, 205
$+ 168-169, 186
$- 168-169, 186
$< 153, 205
$? 153, 205
$@ 153, 205
$[form 154, 171, 176
$[& 154, 171
$[* 154, 171
$[. 154, 171
$[: 154, 171
$[@ 154, 171
$] form 154, 171, 176
$]& 154, 171
$]* 154, 171
$]. 154, 171
$]: 154, 171
$]@ 154, 171
$^ form 154, 171, 176
$^& 154, 171
$^* 154, 171
$^. 154, 171
$^: 154, 171
$^@ 154, 171
WSTRIP 143
command line format 143
WSTRIP command 68
WTOUCH 149, 152, 196, 207
command line format 207
WTOUCH command line 207

invoking WTOUCH 207
WTOUCH options 208

Z

zero flag 232

WATCOM